

Supporting Disciplined Reuse and Evolution of UML Models

Tom Mens, Carine Lucas, Patrick Steyaert[†]

Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2 - 1050 Brussels - BELGIUM
{ tommens | clucas | prsteyae }@vub.ac.be

Abstract. UML provides very little support for modelling evolvable or reusable specifications and designs. To cope with this problem, the UML needs to be extended with support for reuse and evolution of model components. As a first step, this paper enhances the UML metamodel with the “reuse contract” formalism to deal with evolution of collaborating class components. Such a formal semantics for reuse allows us to detect evolution and composition conflicts automatically.

1 Introduction

During the last two decades an entire range of mechanisms has been developed to support the definition, customisation and reuse of implementation-level components such as classes and objects. Some indicative examples are inheritance [13], late-binding polymorphism, object-oriented frameworks [14], meta-object protocols [4] and aspect-oriented programming [6]. Although studies show that significantly more benefits can be gained from reuse during the analysis and design phase than during the implementation phase [3] there is much less support for, and understanding of, reuse at these phases of the software life cycle. Some support for model reuse does exist. Examples are “facades” and “variation points” [2] and “synthesis” of role model components [11]. In general, however, it is not clear what an analysis or design component is, and even less clear how such model components can be composed and reused.

Taking UML 1.1 [10] as a representative example, we observe that it does not provide enough support for dealing with reusable and evolvable model elements. To go beyond the reuse of single classes, packages can be used to encapsulate model elements and pattern structures and templates to define generic models. Experience with implementation reuse has learned, however, that besides the issue of encapsulation, customisation and composition of complex models needs to be

[†]This research was made possible by a research grant of the Flemish Institute for Promotion of Scientific and Technological Research in the Industry (IWT)

addressed as well. Unfortunately, customisation of packages containing arbitrary nested model elements is poorly supported in UML.

Experience with implementation reuse has also learned that building and reusing components is best supported by an iterative process. The reuser can only gain insights in the qualities of reusable components by reusing them in new applications. The provider can only improve the qualities of components if the experience of reuse is fed back to him. Unlike what is sometimes believed, this iteration must be sustained beyond the initial iterations for making reusable components [1]. Successful components can have a long life-span and thus need to evolve and adapt to new reusers and new requirements. The inability to do so turns a reusable component into a legacy component. To be able to sustain the iteration that underlies successful reuse, the issue of how reusers can be supported in upgrading their applications to improved components, must be addressed. Therefore, we introduce *disciplined reuse* as a form of reuse where a maximal degree of consistency is maintained between reusable components and the systems in which they are reused.

In the absence of disciplined reuse, the provider of a component cannot easily benefit from the improvements made by a reuser or from the knowledge that is gained by reusing the component. The reuser does not benefit from improvements made to the component afterwards, nor from the improvements made by other reusers. Moreover, non-disciplined reuse leads to serious maintenance and version management problems. No consistency exists between the model used by the provider and the modified model employed by the reuser. So both models need to be maintained separately, ultimately leading to a version proliferation: the more reusers, the more versions there are, and the less it is clear which version contains which feature, and which part of which version was reused in which other version.

With few exceptions, the most widespread form of reuse at analysis and design level is *copy-and-paste reuse*. With this kind of reuse the reuser takes a copy of a model and changes it to new requirements without maintaining any form of consistency with the original model (so, it is not disciplined at all). Obviously, copy-and-paste reuse of analysis and design models as practised today is not adequate for the needs of organisations that want to employ reuse in a systematic way. Today, organisations are investing in corporate-wide models, in order to be able to reuse them in their applications. Even more, industry-wide initiatives such as OMG are defining models that can be reused by large sections of the industry. The idea of such models is that they are shared by a large number of people over a long period of time, thereby giving a maximum return on investment but also a maximum potential for applications that are open towards each other (because they share a common model). For this to become reality, these models must not only be *shared* – rather than copied – by different reusers, but the model itself must be able to *evolve*, while keeping reusers consistent with it.

The remainder of this paper is outlined as follows. Section 2 discusses the basic concepts and terminology for reuse and evolution of model components, and provides a framework for thinking about model reuse in general. Section 3 presents the reuse contract formalism as a notation and semantics for dealing with modification of UML model elements, and shows how it can be incorporated in the UML metamodel. More

specifically, we discuss how to encapsulate collaborations of class interfaces in model components and how to reuse them. The ideas are based on our previous work on *reuse contracts*: in [12] reuse contracts were defined as a means for maintaining the consistency between an evolving parent class and its subclasses, and in [7] this idea was extended and formally defined for collaborating classes. Section 4 explains how this reuse contract formalism allows us to detect conflicts during evolution, composition and reuse of model components. Finally, section 5 concludes and discusses some future work.

Although we will not discuss practical applications in this paper, the inspiration for what is presented comes from very practical problems, such as how to maintain the consistency in a family of analysis and design models, and how to deal with reuse and evolution as early as possible in the life-cycle. This is also the reason why UML was chosen to express our ideas, as UML has become a standard modelling notation.

2 Reuse and Evolution of Model Components

2.1 Components are the Units of Reuse

The term *component* is interpreted very broadly by most members of the OO community. A component can be a single class, a library of classes, a set of objects that collaborate, a fully fledged framework, and so on. It can be a piece of code, or a model, or even a combination of both. The common characteristic is that *components are the units of reuse*. Obviously, what is considered a unit of reuse heavily depends on the kind of reuse technique that is adopted: with a copy-and-paste reuse mechanism any part of an object model can be considered a component. Since we are only interested in disciplined forms of reuse we will restrict ourselves to more coherent components. Moreover, we will only look at *model components* here, i.e. components defined at the modelling level only.

In this paper we focus on model components consisting of collaborating class interfaces. We have chosen for this kind of model components because collaborations are good building blocks for modelling object-oriented application families. In fact, collaboration patterns can be seen as the modelling-level equivalent of object-oriented frameworks. Their static aspects are represented semantically in UML by means of *collaborations*. At specification level, a collaboration describes the entities (called the *classifier roles*) that participate in the collaboration, their interfaces, and the relationships between these participants (described by *association roles*). At instance level, the collaboration presents instances and links conforming to the classifier roles and association roles in the specification. Finally, the dynamic aspects of the collaboration are represented by *interactions*. These describe the object interaction or message sending behaviour. In UML, two equivalent kinds of interaction diagrams are distinguished: sequence diagrams and collaboration diagrams.

2.2 Incremental Component Modification

Most components cannot be reused as-is, but need to be adapted or customised for reuse. The simplest way to modify a component is by editing the component itself. This has obvious deficiencies: it is not clear how the component is adapted, and the original component is no longer available afterwards. So, in general, an *incremental modification* mechanism is preferred. With incremental modification a modified component is obtained by composing an existing component (the one that is to be modified) with a component modifier (the modification) through some modification mechanism.

Incremental modification has been explored mostly at the programming level with inheritance being the most wide-spread incremental modification mechanism [13]. At the analysis and design level the notion of incremental modification is less well developed. A number of different ways to express relationships between modelling elements in UML exists. The *generalisation* relationship can be used to specify incremental modification, but only to *add* more information (since the more specific element needs to be *substitutable* with the more general element). This is too restrictive for our purposes. An alternative is to use the *dependency* relationship. This is a common mechanism that can be used to indicate a situation in which a change to the target element (the “supplier”) may require a change to the dependent source element (the “client”). In the UML metamodel, the dependency relationship, depicted by a dashed arrow from client to supplier, is specialised to four different relationships: “refinement”, “usage”, “trace” and “binding”. Unfortunately, these relationships are not directly suitable to express reuse or evolution. “Binding” is a relationship between a template and its instantiation, “refinement” is a relationship between modelling elements at different levels, “trace” is only a conceptual connection between modelling elements, and “usage” relates different modelling elements that require each others use. Because we want to use the dependency relationship for expressing evolution, we need to define our own specialisation of the dependency relationship.

2.3 Evolution and Composition Conflicts

As discussed in the introduction, developing reusable components is an iterative process. It is therefore very important that components can evolve. However, component evolution involves a certain cost: all reusers must consider upgrading to the new version and eventually must actually upgrade. Evolution, also, may cause unexpected behaviour in reusers. A reuser that upgrades to a new version of a component can experience different problems: the behaviour of the evolved component has changed, properties of the component that were valid before do not hold anymore, and so on. This kind of conflicts is referred to as *evolution conflicts*.

Furthermore, a component that is reused improperly may cause unexpected behaviour, both in the reuser and in the component itself. Or, even worse, two components that exhibit correct behaviour when reused separately may cause errors

when reused both together in the same system. These kinds of conflicts are called *composition conflicts*.

Conflicts show up during evolution or composition because properties that were relied on by reusers have become invalid. At the programming level composition and evolution conflicts result in erroneous or unexpected behaviour [5, 12]. From a modelling perspective, composition and evolution conflicts may result in a model that is inconsistent (for example, referencing model elements that do not exist anymore), or in a model that does not have the meaning intended by the different reusers. An example will be given in section 4.

3 Reuse Contracts

In the rest of this paper we show how the formalism of reuse contracts [7, 12], which is used to support disciplined reuse and evolution of model components, can be incorporated into the UML language by directly extending its metamodel. We will basically follow the same approach as in the UML semantics document: the abstract syntax is given by means of UML class diagrams, while additional well-formedness rules and constraints are expressed in semi-formal natural language. We have deliberately chosen *not* to express the constraints in OCL because the fact that OCL does not have a complete formal semantics leads to many problems, ambiguities and open questions [9, 15]. Moreover, there is virtually no support for OCL in existing CASE tools that have adopted the UML notation.

3.1 Informal Discussion

The idea behind *reuse contracts* is that a component is reused on the basis of an explicit contract between the *provider* of the component and a *reuser* that modifies this component. The purpose of a contract is to make reuse more disciplined. For this purpose, both the provider and the reuser have obligations. The primary obligation of the provider is to document how the component *can be* reused. The reuser needs to document how the component *is* reused or how the component evolves. Both the provider's and reuser's documentation must be in a form that allows to detect what the impact of changes is, and what actions the reuser must undertake to "upgrade" if a certain component has evolved. To summarise we can say that a reuse contract helps in keeping the model of the provider consistent with the model of the reuser.

Before the provider can document evolution, he needs to document what properties of the component can be relied on at a particular point in time. The *provider clause* states certain properties of the entities in the provided component. In the *reuser clause*, the reuser documents the changes made to the provided component. The *contract type* expresses *how* the provided component is reused. Possible contract types include extension, cancellation, refinement and coarsening. The contract type imposes obligations, permissions and prohibitions onto the reuser. For example, the extension contract type obliges reusers to add new elements, but prohibits overriding of existing

elements. It permits adding multiple elements at once. Contract types and the obligations, permissions and prohibitions they impose are fundamental to disciplined reuse, as they are the basis for detecting conflicts when provided components evolve.

In a reuse contract, the provider clause provides only a certain view on the component. Thus a component can participate in different reuse contracts that address different concerns of the provided component. Typical examples of general concerns are persistence, distribution and user interaction.

3.2 Provider Clause Notation

In order to deal with model components consisting of collaborating class interfaces, the provider clause must be a stereotyped «provider clause» package consisting of two parts: a collaboration specification and a set of interactions owned by the collaboration. Their corresponding diagrams are encapsulated in stereotyped packages with stereotypes «collaboration» and «interaction» respectively², as illustrated in the left part of Fig. 1. Such an abstract collaboration between template class interfaces can be instantiated to an actual collaboration between existing classes by using the so-called *pattern* notation. This pattern fills in the different roles specified in the collaboration with links to actual classes (or class interfaces, that hide the implementation details of the actual class). As a result, these classes must satisfy the behaviour imposed by the collaboration. The notation for this is shown on the right in Fig. 1.

The example of Fig. 1 represents part of the design for navigation in a webbrowser. There are only two important participants: `Browser` and `Document`. These classifier roles can communicate with each other over two unidirectional association roles: `browser` and `doc`. The `Document` interface contains two operations: `mouseClick` describes what happens when the mouse is clicked in some part of the document, and `resolveLink` expresses what happens when a hyperlink is followed in the document. The `Browser` interface also contains two operations that are important for navigation: `handleClick` and `getURL`.

² We are aware of the fact that “collaboration” and “interaction” are reserved keywords, but it is the most obvious name to choose for a stereotyped package containing a *Collaboration* and an *Interaction* respectively.

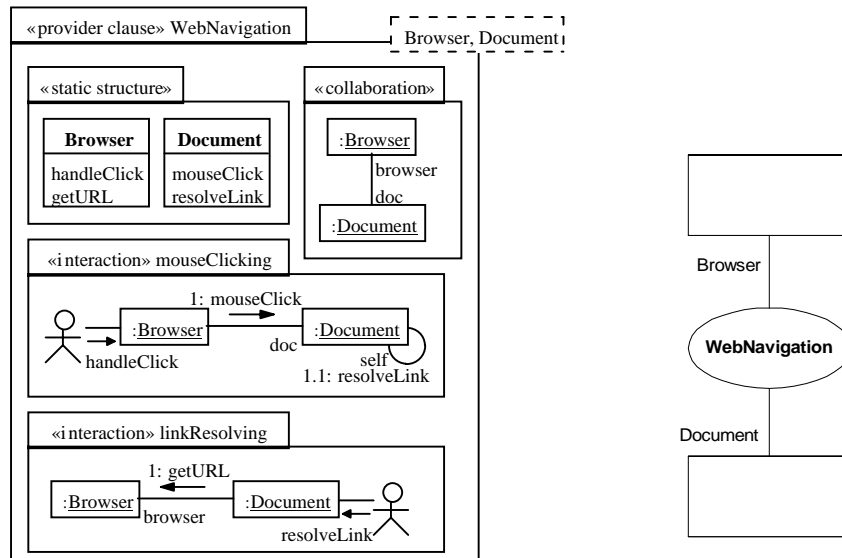


Fig. 1. Provider clause notation for the *WebNavigation* component, and pattern notation for using this component as a collaboration between actual classifiers (which can be classes or class interfaces). To stress the fact that *Browser* and *Document* are only template roles that need to be filled in before use, the template notation (a dashed rectangle) is used.

There are two interactions corresponding to the collaboration. In *linkResolving* part of the navigation behaviour is made explicit: when `resolveLink` is invoked in an instance of the *Document* participant, a message `getURL` will be sent to an instance of the *Browser* participant to fetch the contents of the web page pointed to by the hyperlink. The *mouseClicking* interaction describes what happens if a mouse click is detected by the browser. When this click occurs inside a document, `handleClick` sends a `mouseClick` message to an instance of *Document*, which determines if this mouse click causes a link to be followed. If this is the case, the `resolveLink` self send is issued³.

Fig. 2 shows how the *WebNavigation* collaboration pattern can be used to express an actual collaboration between different classes in a class diagram representing the design of an object-oriented application framework. Different collaboration patterns can be used to indicate the many different roles that classes play (in the figure we have mentioned only three: *Printing*, *WebNavigation* and *PDFNavigation*), and the same pattern can be used in different places of the diagram, with different classes filling in the roles of the collaboration pattern.

³ In the *mouseClicking* interaction an invocation is performed along a self link that was not explicitly modelled as an association. We assume that the self link is always implicitly present, but only show it when an invocation is modelled along it.

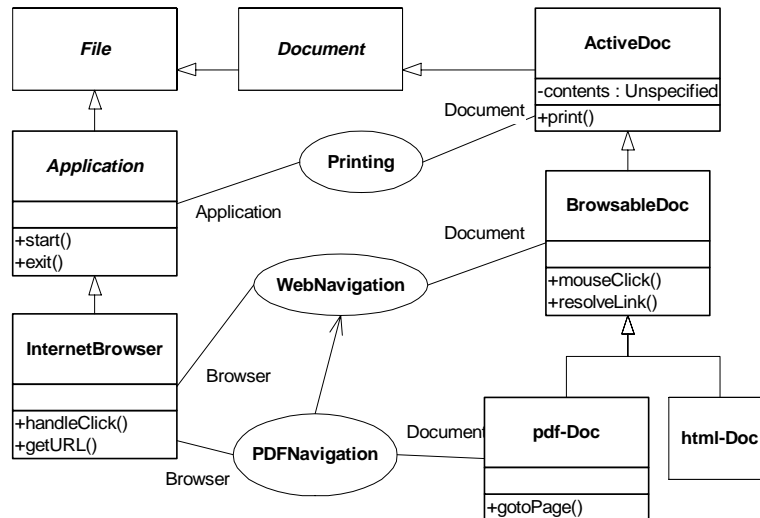


Fig. 2. “Instantiation” of some collaboration patterns (*Printing*, *WebNavigation* and *PDFNavigation*). The Browser role of the *WebNavigation* collaboration is filled in by the *InternetBrowser* class, and the Document role is filled in by the *BrowsibleDoc* class. Sometimes collaborations can depend on other ones. For example, *PDFNavigation* depends on *WebNavigation* (as will be shown in Fig. 5).

3.3 Provider Clause Semantics

In the UML metamodel of Fig. 3, the *Collaboration* metaclass expresses which entities participate in the reusable component. Each participant, called *ClassifierRole*, is semantically represented by an *Interface* that has a *name* and owns an ordered set of *Operations*. The relation between the different *ClassifierRoles* is specified by means of *AssociationRoles*. The *Collaboration* owns a set of *Interactions* (encapsulated in «interaction» packages) that describe how instances of the different *ClassifierRoles* interact by means of message sends. Many different *Interactions* can correspond to the same *Collaboration*. They can be distinguished by their package name. Similarly, a *ClassifierRole* can have many different instances that are distinguished by their object names. Schematically, the abstract syntax corresponding to the description above is given in Fig. 3. A *ProviderClause* is defined as a specialisation of *Package*. The part about *Collaborations* and *Interactions* is taken over from the UML metamodel.

There are many constraints that need to be fulfilled by the different elements of Fig. 3. For example, consistency needs to be maintained between all *Interactions* and the *Collaboration* specified in the provider clause. Fortunately, many of these

consistency constraints are already defined in the UML semantics. Since we are dealing with reuse contracts, we need to impose a number of additional requirements⁴:

1. Each *Interaction* corresponding to a certain *Collaboration* must be owned (indirectly) by the same *ProviderClause*.
2. The *base* of each *ClassifierRole* must be an *Interface*. Consequently, each *availableFeature* in a *ClassifierRole* must be an *Operation*.
3. The *specification* of each *Message* in an *Interaction* must be an *Operation* that is an *availableFeature* in the *ClassifierRole* which is the *receiver* of the *Message*.
4. The *specification* of the *activator* of each *Message* must be an *Operation* that is an *availableFeature* in the *ClassifierRole* which is the *sender* of the *Message*.
5. An *Interaction* can only contain a *Message* if its *sender* and *receiver ClassifierRoles* are connected by means of an *AssociationRole* in the *Collaboration* which is the *context* of the *Interaction*.

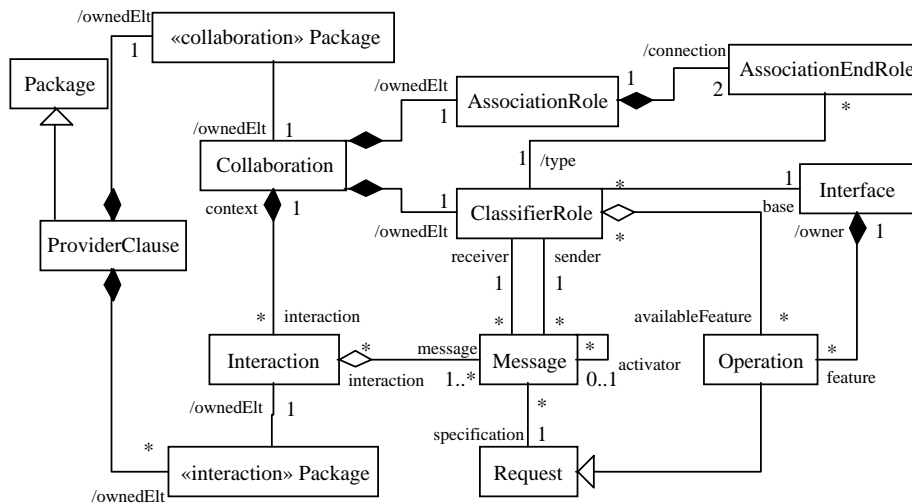


Fig. 3. Collaborations and interactions in UML metamodel

A final issue that is not covered in the UML semantics is how *Interactions* can be kept mutually consistent. In our view, when two or more *Interactions* correspond to the same *Collaboration*, they need to describe independent behaviour. In other words, the same behaviour should not be written twice in different *Interactions*, and the same operation may not exhibit different behaviour in different *Interactions*. This can be guaranteed by prohibiting each operation to play the role of sender in more than one *Interaction*⁵. In other words, all messages sent by the same operation must be shown in the same *Interaction*. For example, the interactions `linkResolving` and

⁴ Some of these well-formedness rules have been given in mathematical notation in [7] and [8].

⁵This rather strong restriction could be weakened, but we have chosen not to do this here for the sake of simplicity.

mouseClicking of Fig. 1 are independent. The linkResolving interaction only describes the behaviour of resolveLink, while mouseClicking only describes the operations invoked by handleClick and mouseClicked.

3.4 Reuser Clauses

Besides a provider clause, a reuse contract also contains a *reuser clause* that describes the incremental modifications that are made to the provider clause. The 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 by using a tag-value pair with tag `modification` that can have two values: `added` and `removed`. A modelling element that is annotated with `{modification=removed}` will be removed in the reused model. If it is annotated with `{modification=added}` it will be added in the reused model. In the rest of this paper we will use the abbreviation `{removed}` and `{added}`. An example is given in Fig. 5.

Similar to *ProviderClauses*, *ReuserClauses* are defined in the extended UML semantics as a specialisation of *Package*, and are denoted graphically by means of a «reuser clause» stereotype. Since reuser clauses enumerate the changes only, they contain only partial information. As a result of this, some of the well-formedness rules that were needed for provider clauses are no longer needed for reuser clauses. The structural part of the semantics of reuser clauses as well as provider clauses is shown in Fig. 4.

3.5 Contract Types

The contract type is an annotation on the reuse contract that expresses in which way the reuser clause incrementally modifies the provider clause. In the UML metamodel, contract types are specified by attaching an attribute with name *contractType* to the reuse contract (see Fig. 4). The value of the *contractType* attribute imposes constraints on the reuse contract in which it occurs, in the sense that the reuser clause must satisfy the requirements specified by the contract type. For example, a reuser clause corresponding to a contract type “interaction refinement” is only allowed to add operation invocations, not to remove them. See [7] for a more detailed discussion of all possible contract types and the constraints they impose.

In Table 1 we have presented a basic set of orthogonal contract types. Together with the constraints they impose, they describe the primitive modifications that can be made to a provider clause. Observe that these basic contract types are sufficient to model all changes to our current model. When new kinds of model elements are added, however, extra contract types might be required. One example from [7] is the addition of contract types “participant concretisation” and “participant abstraction” when an annotation abstract or concrete is added to operations.

Table 1. Basic set of contract types

Contract type	Meaning	Constraint
collaboration extension	adding classifier roles to the collaboration	new classifiers must have a name that differs from existing ones
collaboration cancellation	removing classifier roles from the collaboration	the classifiers should not be referred to in the interactions
collaboration refinement	adding association roles between classifier roles	new association roles must have a name that differs from existing ones
collaboration coarsening	removing association roles from the collaboration	there should be no message sends over this association role in the interactions
participant extension	adding operations to classifier roles	new operations must have a name that differs from existing ones
participant cancellation	removing operations from classifier roles	the operations should not be referred to in any of the interactions
interaction extension	adding instances of classifier roles to an interaction	the added instances must correspond to an existing classifier in the collaboration
interaction cancellation	removing instances of classifier roles from an interaction	there should be no operation invocations to or from these instances
interaction refinement	adding operation invocations to an interaction	there should be an association role in the collaboration over which the invocation can take place, and the operations should be present in the classifier roles
interaction coarsening	removing operation invoc. from an interaction	<i>no constraint</i>

3.6 Reuse Contracts

In the extended UML metamodel shown in Fig. 4, a *ReuseContract* is modelled as a specialisation of a *Dependency* relationship between two modelling elements. The *supplier* of the *ReuseContract* must be a *ProviderClause*, while the *client* of the *ReuseContract* must be a *ReuserClause*. Moreover, the *ReuseContract* must contain a *contractType* attribute.

The complete definition of *BasicProvider* in Fig. 4 has been given earlier. The definition of *BasicReuser* is similar, except that it is not necessarily well-formed, and that its elements may be tagged with the value *added* or *removed*. Also, a *BasicReuser* does not require the presence of both collaboration and interactions. Only the diagram that is subject to modification needs to be mentioned. The parts in Fig. 4 that deal with composite providers and composite reusers are explained in the following subsection.

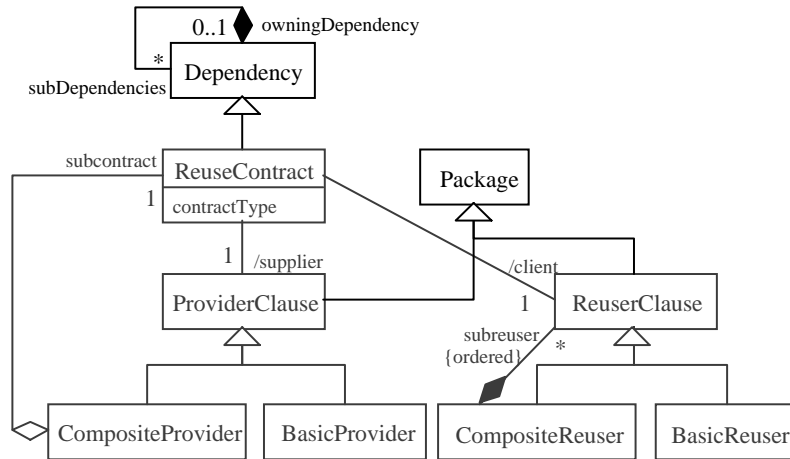


Fig. 4. Extension of the UML metamodel with reuse contracts

3.7 Composite Reuse Contracts

The difference between a model component and its reused version can be quite large. Complex reuser clauses might be necessary to describe this difference. In order to reduce their complexity, and to increase the understandability of reuser clauses, we introduce *composite reuser clauses*. A *CompositeReuser* is simply an ordered sequence of reuser clauses, which can be composite reuser clauses again. The only restriction is that no cycles are introduced, i.e. a composite reuser clause cannot contain itself (either directly or indirectly). The order of the subclauses is important, since the provider clause that is modified by the composite reuser clause will be incrementally modified by applying the subclauses in the specified order.

To be able to deal with composite reusers, we also need a notion of composite reuse contracts. For this we can make use of the fact that in UML a *Dependency* is allowed to have any number of *subDependencies*. With this in mind, we can define a composite *ReuseContract* as a dependency between a *ProviderClause* and a *CompositeReuser* clause, and this dependency contains as many subdependencies as there are subclauses in the composite reuser clause. Reuse contracts can thus be defined at different levels of granularity.

Fig. 5 depicts a composite reuse contract that incrementally modifies the *WebNavigation* provider clause of Fig.1 with a composite reuser clause *PDFNavigation*. The idea is to introduce a new kind of document that 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 itself. This is achieved by a composite reuser clause with three different subclauses. The first one adds an operation *gotoPage* to the *Document* participant, the second one removes the operation invocation from *resolveLink* to *getURL*, and the last one adds an operation invocation from

resolveLink to gotoPage. Each of these reuser clauses are part of a reuse contract, respectively with contract type “participant extension”, “interaction coarsening” and “interaction refinement”. These three reuse contracts are in their turn subdependencies of a larger composite reuse contract with contract type “composite”.

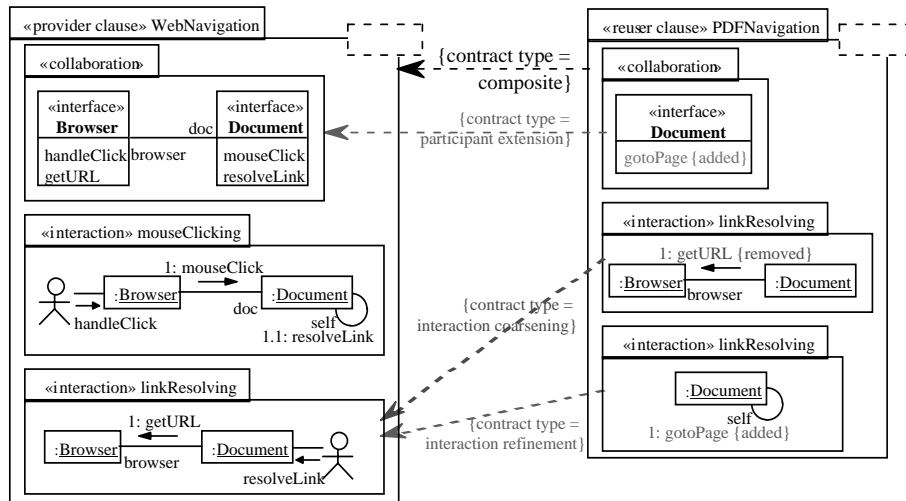


Fig. 5. Example of a composite reuse contract that modifies the *WebNavigation* provider clause to a *PDFNavigation* collaboration. In an actual class diagram, both collaborations and the relation between them can be instantiated using the pattern notation and dependency relationship, as illustrated in Fig. 2.

A reuse contract can be applied to a *BasicProvider*, as in Fig. 5, but also to a reuse contract itself, by wrapping it inside a *CompositeProvider*. This is for example needed to deal with successive incremental modifications at different points in time.

To deal with composite reuse contracts, the metamodel presented in Fig. 4 has to satisfy the following well-formedness rules:

1. The *subreuser* relationship (on *ReuserClause*), the *subcontract* relationship (on *ReuseContract*) and the derived *subDependencies* relationship (on *ReuseContract*) should not introduce any cycles.
2. If a *ReuseContract* has *subDependencies*, then its *client* must be a *CompositeReuser* whose *subreusers* are *clients* of each *subDependency* of the original *ReuseContract*. Moreover, the *suppliers* of each *subDependency* must be owned by the *supplier* of the *ReuseContract*.
3. If the *contractType* of a *ReuseContract* is any of the basic contract types of Table 1, then the *client* is a *BasicReuser*. If the *contractType* is something else, then the *client* is a *CompositeReuser*.

4 Reuse Conflicts

4.1 Evolution and Composition Conflicts

Fig. 6 presents two seemingly orthogonal modifications of the *WebNavigation* provider clause. The first one is the *PDFNavigation* reuser clause of Fig. 5. The second reuser clause, *HistoryNavigation*, adds history functionality to the original webbrowser. Each time a hyperlink is followed through `getURL`, the URL of this link is stored somewhere (by invoking `addURL`), to be able to return to this location at a later time.

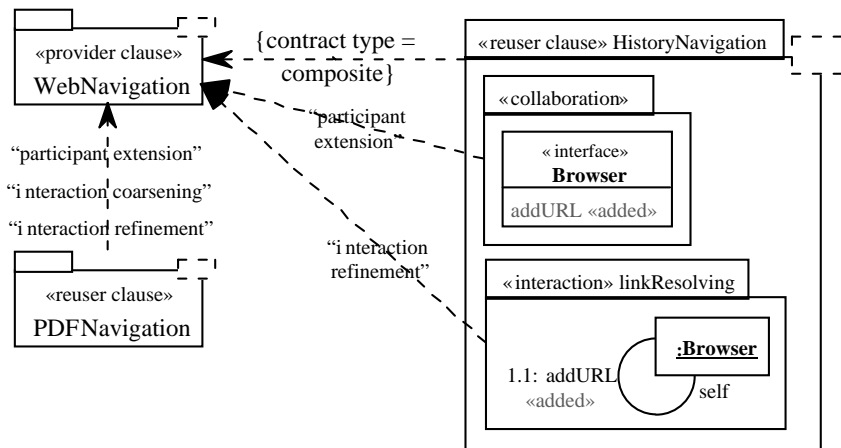


Fig. 6. Detecting a conflict between two modifications of *WebNavigation*.

Both modifications work fine separately, but when they are combined a conflict arises. Since link resolving is done 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 `Document`. This is called the *inconsistent operations* problem. It occurs when one modification adapts a certain operation (here `getURL`) assuming that other operations invoke it, while another modification removes one of these invocations (here the invocation of `getURL` by `resolveLink`). Inconsistent operations can only appear when operation invocations are removed. This can only be achieved by a reuse contract with contract type “interaction coarsening”. For the conflict to occur, the second reuser clause should change the operation from which the invocation is removed, so it can only be a reuse contract with contract type “interaction refinement” or “interaction coarsening”.

In general, problems can occur when two independent changes are made to one model, regardless of whether this is achieved through composition, during evolution or by different developers. We therefore try to detect conflicts between two reuse contracts with the same provider clause, as this models two modifications of the same

model component. There are different approaches possible to detect the conflicts. In most cases, conflicts can be detected by comparing the two contract types and reuser clauses. The inconsistent operations conflict is an example of this. Sometimes, however, the provider clause needs to be consulted or the result of applying the modifications needs to be computed. This is the case for conflicts that involve the transitive closure of operation invocations. [7] inventorises a number of conflicts that can occur and describes how reuse contracts aid in detecting them.

For each conflict a formal rule can be set up to detect the conflict. 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 determining under which conditions they interact in an undesired way. Using these tables it becomes possible to detect a number of conflicts automatically.

One should note that we are not able to detect all possible conflicts between different modifications of the same provider clause. In order to detect some conflicts, more behavioural information is needed. This is one of the trade-offs we made when developing the reuse contract approach. It should detect as much conflicts as possible, yet it should remain intuitive in use in order to be directly usable by software developers.

The approach of detecting conflicts is of course not restricted to basic reuser clauses: it is also scalable to composite reuse contracts and reuser clauses. This is crucial to the usefulness of our approach, as real modifications usually require composite reuser clauses. In general, the conflicts caused by composite reuse contracts can be detected by considering the basic reuse contracts from which they are made up two by two. By first performing some transformations on the composite reuser clauses we can even prohibit the detection of too many conflicts. For example, if a “participant extension” with a certain operation is followed by a “participant cancellation” of the same operation, the conflicts caused by the extension should not be considered. Even more important is that by explicitly declaring composite reuse contracts, the reuser gives extra information about how the provided component is reused. This information can lead to more detailed feedback on possible conflicts. This discussion is however outside the scope of this paper.

4.2 Instantiation Conflicts

Since provider clauses represent reusable components (in our case template collaborations between class interfaces), they are not stand-alone, but interact with the environment in which they are used (i.e., instantiated). For example, in Fig. 2 we have seen how the *WebNavigation* and *PDFNavigation* provider clauses can be instantiated to a collaboration between actual classes in a class diagram. This instantiation can lead to a new kind of conflicts called *instantiation conflicts*. For example, when the `mouseClick` operation would be removed by editing the `BrowsableDoc` class, this would result in a conflict with the *WebNavigation* instantiation, since *WebNavigation*

requires the presence of a `mouseClick` operation in its `Browser` role. Even worse, due to the change in `BrowsableDoc`, similar conflicts might arise in all its subclasses because of the inheritance mechanism. In this case, a conflict will also arise in `pdf-Doc`, since `pdf-Doc` plays the role of `Document` in the *PDFNavigation* instantiation, and *PDFNavigation* also requires the presence of the `mouseClick` operation.

Nevertheless, by using provider clauses to document reusable components, and by carefully specifying how these components are reused, it becomes possible to detect instantiation conflicts too.

5 Conclusion and Future Work

This paper introduced a framework for reasoning about definition, modification and reuse of model components consisting of a collaboration specification and a set of associated interactions. We described how to achieve disciplined reuse and evolution of these components by modelling incremental modification by reuse contracts. Such formal underpinnings for component reuse allow us to introduce rules for conflict detection upon evolution, composition and reuse.

In concreto, we have extended the UML metamodel and introduced a notation to express reuse and evolution of UML model components. Packages offered us a module mechanism to encapsulate the internal details of the components. The dependency relationship between packages was used to document incremental modification of model components. A contract type attribute was attached to this relationship to express different kinds of incremental modification. To allow reuse and evolution to take place at different levels of abstraction (cf. composite provider and reuser clauses) the nesting facility of packages was exploited, together with the possibility of dependencies to have subdependencies.

How easy our ideas can be integrated in a CASE tool that supports UML, will depend on the extensibility of the tool, and of the availability of a metamodel that can be directly accessed and modified from within the tool. Also, a CASE tool can assist in making the approach more user-friendly. We have already done some very promising experiments with adding reuse contracts to Rational Rose by means of its built-in scripting language.

One of the current shortcomings is that the notion of incremental modification, or more generally, the notion of reuse and evolution of components, needs to be defined manually for each kind of component, and for each kind of UML diagram. Another shortcoming is that it is not clear how different kinds of UML diagrams are related to one another. In this paper, we only gave a solution for components containing a collaboration and a set of associated interactions. The same approach could also be used for dealing with components consisting of use case diagrams, state diagrams, deployment diagrams, etc. In a forthcoming paper we will show how the UML metamodel can be extended to deal with evolution of all kinds of UML diagrams in a uniform way.

Another challenge is to use reuse contracts to describe dependencies between components in different phases of the software life-cycle. The transition from model components in one phase to model components in another phase could be explicated by means of reuse contracts and contract types. This should lead to a better traceability between levels, and should enable assessing the impact of making changes to higher level components on the associated lower level components, but also about how modifying lower level components makes them drift away from components at the higher level.

Finally, we are currently applying this approach with an industrial partner, to deploy a set of domain-specific models in different projects. This will provide us the crucial feedback to ameliorate our framework and make it a true methodology for disciplined reuse and evolution.

Acknowledgements

We thank Theo D'Hondt for supporting our work, Stephane Ducasse for the fruitful discussions, and of course Kim Mens, Koen De Hondt and Roel Wuyts for the great collaboration. We also express our gratitude to the anonymous referees for reviewing our paper and providing some valuable feedback.

References

1. Codenie, W., De Hondt, K., Steyaert, P. and Vercammen, A.: From Custom Applications to Domain-Specific Frameworks. Communications of the ACM, Special Issue on Application Frameworks, 40(10). ACM Press (1997) 70-77.
2. Jacobson, I., Griss, M. and Johnson, P.: Software Reuse, Architecture, Process and Organization for Business Success. ACM Press (1997).
3. Karlsson E.-A.: Software Reuse - A Holistic Approach. Wiley (1995).
4. Kiczales, G., des Rivières, J. and Bobrow, D.G.: The Art of the Meta-object Protocol. MIT Press (1991).
5. Kiczales, G. and Lamping, J.: Issues in the design and documentation of class libraries. Proceedings of OOPSLA '92, ACM SIGPLAN Notices, 27(10). ACM Press (1992) 435-451.
6. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J.: Aspect-oriented Programming. Proceedings of ECOOP '97, LNCS 1241. Springer-Verlag (1997) 220 - 242.
7. Lucas, C.: Documenting Reuse and Evolution with Reuse Contracts. PhD Dissertation. Vrije Universiteit Brussel (1997).
8. Mens, T., Lucas, C. and Steyaert, P.: Giving Precise Semantics to Reuse in UML. Proceedings of ICSE '98 Workshop on Precise Semantics for Software Modeling Techniques, Technical Report TUM-I9803. Technische Universität München (1998) 73-89.
9. Gogolla, M. and Richters, M.: On constraints and queries in UML. The Unified Modeling Language - Technical Aspects and Applications. Physica-Verlag (1998).
10. Object Management Group: Unified Modeling Language 1.1 Document Set. OMG Documents ad/97-08-01 to ad/97-08-08 (1997).

11. Reenskaug, T., Wold, P. and Lehne, O. A.: Working With Objects. Manning Publications, Greenwich, CT (1996).
12. Steyaert, P., Lucas, C., Mens, K. and D'Hondt, T.: Reuse Contracts - Managing the Evolution of Reusable Assets. Proceedings of OOPSLA '96, SIGPLAN Notices, 31(10). ACM Press (1996) 268-286.
13. Wegner, P. and Zdonik, S. B.: Inheritance as an Incremental Modification Mechanism, or what like is and isn't like. Proceedings of ECOOP '88, LNCS 276. Springer-Verlag (1988) 55-77.
14. Wirfs-Brock, A.: Designing Reusable Designs - Experiences Designing Object-Oriented Frameworks. Addendum to the OOPSLA/ECOOP '90 Proceedings, SIGPLAN Notices Special Issue. ACM Press (1990) 19-24.
15. Hamie, A., Howse, J., Kent, S., Mitchell, R. and Civello, F.: Reflections on the OCL. Proceedings of <<UML>>'98 International Workshop. Springer-Verlag (1998).