Supporting Reuse and Evolution of UML Models

Tom Mens, Carine Lucas, Patrick Steyaert†

Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2 - 1050 Brussels - BELGIUM
Phone: +32 2 629 3306
Fax: +32 2 629 3525
{ tommens | lucas | prsteyae }@vub.ac.be

Abstract
UML provides little support for modelling reusable specifications and designs. To deal with this problem, we enhance UML with support for reuse and evolution of model components (more specifically, collaborating classes). To this extent, we enhance the UML (meta-) model with the “reuse contract” formalism. Among others, this gives us a formal semantics for reuse that allows us to detect reuse and composition conflicts automatically.

Keywords
reuse, evolution, model components, extensibility

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, late-binding polymorphism, object-oriented frameworks [Wirfs-Brock90], meta-object protocols [Kiczales&al91] and aspect-oriented programming [Kiczales&al97]. Although it is generally acknowledged that reuse is much more important during the analysis and design phase than during the implementation phase, 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” [Jacobson&al97] and “synthesis” of role model components [Reenskaug&al96]. 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 [Rational97] as a representative example, we observe that it does not provide enough support for dealing with reusable components. To go beyond the reuse of single classes, packages can be used to encapsulate model elements and pattern structures 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 addressed as well. Unfortunately, customisation of packages containing complicated components is poorly supported in UML.

Experience with implementation reuse has also learned that building and using 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 [Codenie&al97]. Successful components can have a long life-span and thus need to evolve and adapt to new reusers and their 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

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

In this paper we discuss a framework for reasoning about definition, modification and reuse of UML models. We focus on models that are made up of collaborating classes. Such collaborations seem to be good building blocks for modelling object-oriented application families. They are the basis of the modelling-level equivalent of object-oriented frameworks. We discuss how to encapsulate collaborations in model components and how to reuse them. To this extent, we enhance the metamodel of UML, to enable detection of conflicts during evolution and composition of model components. The ideas are based on our previous work on reuse contracts. In [Steyaert&al96] reuse contracts were defined as a means for maintaining the consistency between an evolving parent class and its subclasses. In [Lucas97] this idea was extended and formally defined for collaborating classes (at the implementation level).

Although we will not discuss practical applications in this paper, the inspiration for what is presented comes from very practical reuse problems, such as how to maintain the consistency in a family of analysis and design models, and how to reason about reuse as early as possible in the life-cycle. This is also the reason why UML was chosen to express our ideas, as UML is becoming the standard modelling notation. Another reason is UML’s powerful built-in extension mechanisms such as stereotypes and constraints.

2. Reuse and Evolution of Model Components

This section discusses the basic concepts and terminology for discussing reuse and evolution of model components. It provides a framework for thinking about model reuse in general and gives concrete examples in UML.

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 classes. The static aspects are represented in UML by means of interfaces and collaborations. While the former describes the operations of the entities (called the participants) that participate in the collaboration, the latter describes their relationships with other participants. 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 [Wegner&Zdonik88].

Incremental modification has been explored mostly at the programming level with inheritance being the most wide-spread incremental modification mechanism. 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”). The dependency relationship, depicted by a dashed arrow from client to supplier, may be adorned with an optional stereotype and an optional name. By formally defining our own stereotyped dependencies, we will be able to distinguish between different kinds of incremental modification.
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 [Kiczales&Lamping92, Steyaert&al96]. 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 is given in section 4.

3. Reuse Contracts

In the rest of this paper we show how the formalism of reuse contracts [Steyaert&al96, Lucas97] can be used to support disciplined reuse of model components, and we incorporate this formalism into UML.

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 Clauses

Since we only deal with model components consisting of collaborating classes in this paper, the provider clause will be a stereotyped package (with stereotype «provider clause») consisting of three main parts: a set of interfaces, a collaboration, and a set of associated interactions. Their corresponding diagrams are encapsulated in stereotyped packages with stereotypes «static structure», «collaboration» and «interaction» respectively1, as illustrated in Figure 1. Since this notation is somewhat cumbersome, we prefer to use the alternative notation proposed in Figure 2.

The «static structure» package expresses which entities participate in the reusable component. Each of these participants is semantically represented by an Interface that has a name and owns an ordered set of Operations. The «collaboration» package expresses the different roles each participant plays, and how these roles are related (by means of AssociationRoles). Semantically this is represented by a Collaboration. Finally, each «interaction» package contains an Interaction that describes how the different roles in its associated Collaboration interact. Many different

---

1 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.
Interactions can correspond to the same Collaboration. They can be distinguished by the package name. Similarly, a participant can play many different roles that are distinguished by their role names.

![Figure 1: Stereotyped package notation](image)

The example depicted in Figure 1 represents part of the design for navigation in a web browser. There are only two important participants: Browser and Document. These can communicate with each other by means of 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.

There are two interactions corresponding to the collaboration. In linkResolving part of the navigation behaviour is made explicit: when resolveLink is invoked in a participant role of type Document, a message getURL will be sent to a participant role of type Browser 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 the document, which determines if this mouse click causes a link to be followed. If this is the case, the resolveLink self send is issued.

Figure 2: Alternative notation

Figure 3 graphically depicts how the different parts of a provider clause are related to one another. The part about Collaborations and Interactions is taken directly from the UML metamodel. There are many constraints that need to be fulfilled by the different parts of this diagram. 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. We briefly summarise the most important well-formedness rules in natural language:

1. A Collaboration may only contain ClassifierRoles and AssociationRoles.
2. The activator of a Message must be contained in the same Interaction as the activated Message.
3. The availableFeatures in a ClassifierRole must be a subset of the features in the base classifier.
4. The sender and the receiver of each Message must participate in the Collaboration which defines the context of the Interaction.

---

2 Note that in the mouseClicking interaction an invocation is performed along a self link. This link was not explicitly modelled as an association role. We assume that the self link is always implicitly present, but we only depict it when an invocation is modelled along it.

3 These well-formedness rules have also been defined formally in mathematical notation in [Lucas97] and [Mens&al98]. We have decided not to express them in OCL, since OCL does not have a complete formal semantics and is not powerful enough to express some constraints.
Since we are dealing with reuse contracts, we need to impose a number of additional requirements:

1. Each Interaction corresponding to a certain Collaboration must be owned (through an extra indirection) by the same «providerClause» Package.

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.

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

3.3 Reuser Clauses

Besides provider clauses, reuse contracts also contain reuser clauses. Reuser clauses describe the incremental modifications that are made to the provider clauses. 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 Figure 4, again using an alternative notation similar to the one for provider clauses.

Similar to provider clauses, reuser clauses are modelled as a package annotated with 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.

---

4This rather strong restriction could be weakened, but we have chosen not to do this here for the sake of simplicity.
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 composed reuser clauses. A composed reuser clause is simply an ordered list of reuser clauses, which can be again composed reuser clauses. The only restriction is that no cycles are introduced, i.e. a composed 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 composed reuser clause will be incrementally modified by applying the subclauses in the specified order.

3.4 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 other words, the contract type puts extra constraints on how the provider clause is reused. In UML, contract types are specified as stereotypes of the form «contract type» that are attached to the reuse contract (see Figure 4). These stereotypes impose constraints on the reuse contracts to which they are attached, in the sense that the reuser clause must satisfy the requirements specified by the contract type.

In Table 1 we have presented a basic set of contract types that are as orthogonal as possible. 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 [Steyaert & al. 96] is the addition of contract types «participant concretisation» and «participant abstraction» when an annotation abstract or concrete is added to operations. Other examples are given in section 5.

<table>
<thead>
<tr>
<th>Contract Type</th>
<th>Meaning</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>«static structure extension»</td>
<td>adding participants to the static structure</td>
<td>new participants must have a name that differs from existing ones</td>
</tr>
<tr>
<td>«static structure cancellation»</td>
<td>removing participants from the static structure</td>
<td>the participants should not be referred to in the collaboration or interactions</td>
</tr>
<tr>
<td>«participant extension»</td>
<td>adding operations to participants</td>
<td>new operations must have a name that differs from existing ones</td>
</tr>
<tr>
<td>«participant cancellation»</td>
<td>removing operations from participants</td>
<td>the operations should not be referred to in any of the interactions</td>
</tr>
<tr>
<td>«collaboration extension»</td>
<td>adding participant roles to the collaboration</td>
<td>new participant roles must have a new name, and must correspond to a participant in the static structure</td>
</tr>
<tr>
<td>«collaboration cancellation»</td>
<td>removing participant roles from the collaboration</td>
<td>there should not be any association roles attached to the participant role</td>
</tr>
<tr>
<td>«collaboration refinement»</td>
<td>adding association roles between participant roles</td>
<td>new association roles must have a name that differs from existing ones</td>
</tr>
<tr>
<td>«collaboration coarsening»</td>
<td>removing association roles from the collaboration</td>
<td>there should not be any operation invocation over this association role</td>
</tr>
<tr>
<td>«interaction extension»</td>
<td>adding participant roles to an interaction</td>
<td>the added participant roles must already be present in the collaboration</td>
</tr>
<tr>
<td>«interaction cancellation»</td>
<td>removing participant roles from an interaction</td>
<td>there should not be any operation invocation to or from these participant roles</td>
</tr>
<tr>
<td>«interaction refinement»</td>
<td>adding operation invocations to an interaction</td>
<td>there should be an association role in the collaboration over which the invocation can take place, and the operations should be present in the static structure</td>
</tr>
<tr>
<td>«interaction coarsening»</td>
<td>removing operation invoc. from an interaction</td>
<td>no constraint</td>
</tr>
</tbody>
</table>

Table 1: Basic set of contract types

3.5 Reuse Contracts
In UML, a reuse contract can be modelled as a dependency between two modelling elements. The supplier of the dependency must be a provider clause, while the client of the dependency must be a reuser clause. Moreover, the dependency must have a contract type attached to it.
To be able to deal with composed reusers, we also need a notion of composed reuse contracts. Fortunately, we can make use of the fact that according to the UML semantics a dependency is allowed to have any number of subdependencies. With this in mind, we can define a «composed» reuse contract as a dependency between a provider clause and a composed reuser clause, and this dependency contains as many subdependencies as there are subclauses in the composed reuser clause. Reuse contracts can thus be defined at different levels of granularity.

Figure 4: Example of a reuse contract

As an illustration of this, Figure 4 depicts a composed reuse contract that incrementally modifies the WebNavigation provider clause of Figure 1 with a composed 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 three different reuser clauses. 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 reuse contract with contract type «composed».

Figure 5: UML diagram of reuse contracts

In Figure 5 the abstract syntax of a reuse contract is given in the form of a UML metamodel. An alternative definition can be given using stereotypes, but since this is more cumbersome we prefer the former definition. The entities presented in Figure 5 have to satisfy the following well-formedness rules:

---

Note that extending the UML metamodel directly requires a meta-metamodel, such as the OMG Meta Object Facility.
Using simply inventories a number of conflicts that can occur and describes how reuse contracts aid in detecting them. Sometimes, comparing the clause of each subDependency of the original ReuseContract.

Finally, each specific contract type introduces its own specific constraints on the reuser clause in the reuse contract. For example, the reuser clause corresponding to an «interaction refinement» is only allowed to add operation invocations, not to remove them. For a more detailed discussion we refer to [Lucas97]. The complete definition of BasicProvider in Figure 5 has been given in section 3.2. 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 all three parts (static structure, collaboration and interactions). Only those diagrams that are subject to modification need to be mentioned.

A reuse contract can be applied to a BasicProvider, as in Figure 4, but also to a reuse contract itself, by wrapping it inside a ComposedProvider. This is for example needed to deal with successive incremental modifications at different points in time.

4. Reuse Conflicts

Consider two orthogonal adaptations of WebNavigation depicted in Figure 6. The first one is PDFNavigation as discussed above. The second, 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. 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 through «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 an «interaction refinement» or an «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 resulting provider clause needs to be computed. This is the case for conflicts that involve the transitive closure of operation invocations. [Lucas97] inventories 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 simply comparing all contract types two by two, and determining whether they can interact in an undesired way. Using these tables it becomes possible to detect conflicts automatically.
The approach of detecting conflicts is not only applicable to basic reuser clauses, but is also scaleable to combined reuse contracts and reuser clauses. This is crucial to the usefulness of our approach, as real modifications usually require combined reuser clauses. In general, the conflicts caused by combined 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 combined 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 combined 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.

5. Conclusion and Future Work

In this paper we introduced a framework for reasoning about definition, modification and reuse of UML model components. We first defined components as a combination of a set of interfaces, a collaboration and a set of associated interactions. Second, we described how to achieve disciplined reuse by modelling incremental modification and composition by reuse contracts. Such formal underpinnings for component reuse allow us to introduce rules for conflict detection upon evolution or composition.

As a case study, we have used the UML notation to express reuse and evolution of model components. UML packages offered us a module mechanism to encapsulate the internal details of the components. Dependency relationships between packages were used to deal with reuse and evolution of model components. Contract types were attached to these relationships to express different kinds of incremental modification. To allow reuse and evolution to take place at different levels of abstraction (cf. combined provider and reuser clauses) the nesting facility of packages was exploited, together with the possibility of dependencies to have subdependencies. How easy it will be to integrate our ideas in a CASE tool will depend on the extensibility of the tool or the availability of a meta-object facility for extending the UML metamodel. Also, a CASE tool can assist in making the approach more user-friendly.

One of the current shortcomings is that the notion of incremental modification of components, 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 (very simple) class diagram, a collaboration and a set of associated interactions.

In order for UML to fully support disciplined reuse and evolution, the general framework discussed in this paper still needs to be elaborated on in several ways. First, it should support the full functionality of collaborations and interactions. As a second enhancement, reuse contracts could be applied to the other diagrams like use case diagrams, sequence diagrams and state-transition diagrams. How to incorporate reuse contracts in state-transition diagrams has already been illustrated in [Mens&Steyaert97].

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 at one level to model components at another level could be explicitated 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.

6. Acknowledgements

We thank Theo D'Hondt for supporting our work, Stephane Ducasse for the fruitful discussions, and of course the rest of the Reuse Contracts Team for the great collaboration.

7. References


