Automated Software Engineering 7, 39–59, 2000
© 2000 Kluwer Academic Publishers. Manufactured in The Netherlands.

Automating Support for Software Evolution in UML

TOM MENS tommens@vub.ac.be
THEO D'HONDT tjdhondt@vub.ac.be
Programming Technology Lab, Vrije Universiteit Brussel, Pleinlaan 2—1050 Brussel, Belgium

Abstract. Disciplined support for evolution of software artifacts is important in all phases of the software life-cycle. In order to achieve this support, a uniform underlying foundation for software evolution is necessary. While, in the past, reuse contracts have been proposed as such a formalism in a number of different domains, this paper generalises the formalism, and integrates it into the UML metamodel. As such, support for evolution becomes readily available for many kinds of UML models, ranging from requirements to the implementation phase.

Keywords: software evolution, UML metamodel, reuse contracts

1. Introduction

In recent years, a lot of attention has been paid to the evolution of reusable implementation-level software artifacts. There is, however, much less support for, and understanding of, evolution during the earlier phases of requirements, analysis and design. Some support such as *facades, variation points* (Jacobson et al., 1997) and *change cases* (Ecklund et al., 1996) does exist. In practice, however, few currently existing CASE tools adequately support evolution of analysis and design models. This lack of tool support is partly due to the fact that the modelling techniques themselves do not provide proper support for evolution. As a representative example, UML 1.3 (OMG, 1999) does not adequately deal with evolvable models, as can be concluded from the following paragraph extracted from the UML semantics specification:

> "Whenever the supplier element of a dependency changes, the client element is potentially invalidated. After such invalidation, a check should be performed followed by possibly changes to the derived client element. Such a check should be performed after which action can be taken to change the derived element to validate it again. The semantics of this validation and change is outside the scope of UML."

Automated support for software evolution is central to solving some very important technical problems in current day software engineering. We discuss just a few of these problems below. A first topic is *propagation of changes*. When some part of the software is modified, many other parts of the software need to be changed as well. The identification of these parts and the changes that need to be made to them is not a trivial task. To a certain extent, *impact analysis* techniques have been developed to address this problem (Bohner and Arnold, 1996). A second issue is *consistency maintenance* and *architectural drift*: how

does modification of artifacts during a later phase make them drift away from the artifacts used in earlier phases, and how can this be prohibited by better maintaining the consistency between those artifacts? Other topics as *version control, conflict detection* and *traceability* are also intimately related to the ability of a tool to manage evolution of software artifacts.

In order to provide this support and build CASE tools for it, however, a general underlying foundation for software evolution is needed. While *ease of use* is an important aspect, the approach should be *formal* enough so that it can form a basis for tools, and *general* enough to be applicable in as many domains as possible.

In this paper we propose a general notation and semantics for dealing with disciplined evolution of UML models, based on previous work on *reuse contracts*. Reuse contracts were first introduced in Steyaert et al. (1996) for handling change propagation between a parent class and its subclasses. In Lucas (1997) this idea was extended to deal with evolution of cooperating classes, and Mens et al. (1999) discussed the application of reuse contracts to collaborations in UML. Even the application of reuse contracts to requirements models using Object Behaviour Analysis (Rubin and Goldberg, 1992) has been investigated (D'Hondt, 1998). This paper ties together, extends and generalises these previous efforts by providing one unified approach for software evolution.

In order to fully understand this paper, some knowledge of the UML metamodel and the OCL is required (OMG, 1999). Please note that we make use of version 1.3 of the UML metamodel, which has undergone some significant changes since the first official version 1.1.

2. A unified approach for software evolution

2.1. Evolution conflicts

It is very important that software artifacts can evolve. However, software evolution involves a certain cost: all developers must consider upgrading to the new version of the artifact. Unfortunately, integrating a new version of a software artifact in the context where the old version has been used may cause unexpected interactions because the behaviour of the evolved artifact has changed, or because assumptions that could be made before do not hold anymore. These undesired interactions are referred to as *upgrade conflicts*.

Another, related, conflict occurs when different evolvers independently make changes to the same software artifact. By merging these parallel evolutions into a new version of the software artifact, it is possible that the interaction of both modifications leads to unexpected and undesired behaviour. In that case, we speak of a *merge* or *composition conflict*. At the programming level both kinds of evolution conflicts result in erroneous or unexpected behaviour (Kiczales and Lamping, 1992; Steyaert et al., 1996). On analysis and design level, upgrade and merge conflicts may result in a model that is inconsistent, or in a model that does not have the intended meaning anymore.

As an example of a simple conflict, consider a design model that contains two classes `A` and `B`. One software developer decides to add an association from `A` to `B`, while another software developer independently decides that class `B` should be removed. Clearly, both modifications are incompatible, as their merge would give rise to an inconsistent model

with a dangling association with source `A`. While this is only a structural conflict that can already be detected by some existing merge tools (Westfechtel, 1991), there are also parallel modifications whose merge results in a valid UML model, but with an undesired behaviour. These are the more interesting conflicts, since they cannot be detected by current CASE tools. An illustrative example will be given later in this paper.

In general, conflicts show up during evolution of software artifacts because the assumptions made by dependent artifacts have become invalid. Therefore, the key to detecting conflicts *automatically* is to formally document and monitor these assumptions through an explicit contract.

2.2. *Evolution contracts*

As stated earlier, reuse contracts are a formalism for dealing with reuse and evolution of any kind of software artifact. Because the emphasis of this paper lies on support for evolution, however, we will use the term *evolution contracts* instead. Their generality has already been demonstrated by applying them to different kinds of artifacts: class inheritance hierarchies (Steyaert et al., 1996), cooperating classes (Lucas, 1997), requirements specifications (D'Hondt, 1998), UML collaborations (Mens et al., 1999) and software architectures (Romero, 1999). A formal treatment has been given in the PhD dissertation (Mens, 1999).

The idea behind *evolution contracts* is that incremental modification and evolution of software artifacts is made explicit by means of a formal contract between the *provider* and the *modifier* of the artifact. The purpose of the contract is to make evolution more disciplined. The provider clause specifies what properties of the software artifact can be relied on, e.g., by describing the artifact's interface using IDL, specialisation interfaces (Lamping, 1993), interaction contracts (Helm et al., 1990), collaboration contracts (De Hondt, 1998) or any other formal specification mechanism. Because an essential aspect of evolution contracts is to provide better support for *unanticipated* evolution, the provider clause does *not* specify the admissible changes that can be made to a given software artifact. Instead, the precise way in which the artifact is modified is specified in a separate *modifier clause*. Both contract clauses (provider and modifier) must be in a form that allows us to assess what the impact of changes is, and what actions dependent artifacts must undertake to "upgrade" if a certain artifact evolves. Another important characteristic of evolution contracts is that they only allow us to make changes to an artifact if these changes preserve the consistency (or well-formedness) of the artifact.

In figure 1 an example is given of an evolution contract that expresses the evolution of a `Set` interface into a new version `Set 2`. The provider clause specifies that `Set` contains two operations `insert` and `union`. The modifier clause specifies that two operations `remove` and `intersection` are added.

To classify different kinds of modifications, we make use of *contract types*. A contract type imposes obligations, permissions and prohibitions on the modifier. Contract types and the constraints they impose are fundamental to disciplined evolution, as they are the basis for detecting conflicts when software artifacts evolve. The essence is to find contract types that are specific enough to detect useful evolution conflicts, while remaining general enough

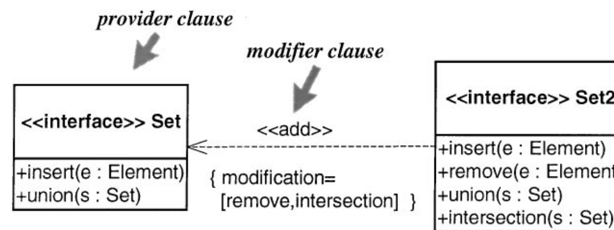

Figure 1. Evolution contract terminology.

to be applicable to many different kinds of software artifacts. Based on our experience with evolution contracts in different domains, we distinguish the following four primitive contract types: *Addition, Removal, Connection* and *Disconnection*. Figure 1 shows an example of *Addition*, designated with stereotype ≪add≫, which allows us to add new model elements (in this case the operations `remove` and `intersection`) to the provider interface, but prohibits overriding or removal of existing model elements. It permits adding multiple elements at once.

A more detailed treatment of the four primitive contract types will be given in Section 4. Section 5 describes how these contract types aid in detecting evolution conflicts. For an in-depth discussion of evolution contracts, using a slightly different terminology, we refer to the PhD dissertations (Lucas, 1997) and (Mens, 1999).

2.3. *The unified modelling language*

Since we are primarily interested in evolution during the early phases of the software life-cycle, we must choose a suitable analysis and design notation in which to express our ideas. Over the years, innumerable analysis and design methods and notations have been proposed and developed. In late 1997, most of these methods have converged to a standard notation, the Unified Modelling Language (UML), which has been accepted as an industry standard by the OMG (Object Management Group, 1999). Besides being a standard, the UML is open, in the sense that new features can be added to it quite easily. For this reason, we use UML notation to express our ideas.

While at first the different incarnations of evolution contracts were developed separately, this paper incorporates evolution contracts in the UML metamodel, thereby making support for evolution available in most of the models currently available in UML, as well as in new kinds of models that might be added to UML in the future. As a result, evolution contracts (and consequently also automated support for evolution) become applicable to UML models in all phases of the software life-cycle, ranging from use case models to deployment models.

3. Incorporating evolution contracts in UML

3.1. *Extension mechanisms, the metamodel and OCL*

An important feature of UML is its openness. First of all, it contains three built-in *extension mechanisms* with which the semantics of existing modelling concepts can be enhanced:

stereotypes, constraints and tagged values. All three mechanisms can be applied to any modelling concept. While these extension mechanisms are suitable for adding user-defined semantics to the UML in many cases, there are some situations where they are not powerful enough. In these situations, it is still possible to extend or modify the UML semantics, since it is (semi-)formally described by means of a metamodel. By directly editing this metamodel, one can alter the UML semantics, at the risk of incompatibility and lack of portability.

The UML semantics is described in the metamodel as a combination of three different views. The abstract syntax of UML concepts is expressed graphically, by using a subset of the UML notation. Well-formedness rules, that describe when instances of a certain language construct are meaningful, are expressed as constraints in OCL. Some remaining constraints have been expressed in natural language because of time pressure.

It needs to be said that the UML semantics is currently not completely formally defined, although there are several ongoing attempts in this direction (Evans et al., 1999; Övergaard and Palmkvist, 1999; Kent et al., 1999). Another problem is that the semantics of OCL itself is not formally defined. As a result, parts of this constraint language lead to problems, ambiguities or open questions (Gogolla and Richters, 1998; Hamie et al., 1999). Recently, however, some attempts have been made in trying to formalise OCL (Richters and Gogolla, 1998), so there is good hope that these problems will disappear in the near future.

In this paper, we enhance UML with support for evolution, with the aim of automatically detecting conflicts between parallel evolutions of the same UML model. Although we try to use the built-in extension mechanisms of UML as much as possible, we sometimes need to extend the UML metamodel as well. All necessary constraints and well-formedness rules will be specified in OCL.

Before we can propose our extension to the UML metamodel, we need to explain some conventions used throughout this paper. When referring to a term of the metamodel in ordinary text, italic font is used, as *ModelElement*. Terms beginning with an uppercase letter refer to metaclasses, while terms beginning with a lowercase letter refer to features or association roles of these metaclasses. The symbol / is used to denote derived elements, i.e., elements that can be directly calculated from other ones. Square brackets [ ] are used to denote a sequence (i.e., and ordered set) of elements.

3.2. ModelElements and relationships

All kinds of elements that can be specified in a UML model are defined as a specialisation of the abstract metaclass *ModelElement* in the UML metamodel. A particularly interesting specialisation of *ModelElement* is *NameSpace*, which is used to represent elements that own other elements. Some concrete specialisations of *NameSpace* are: *Classifiers* (such as *Class, Interface* and *ClassifierRole*) that own a number of *Features* (such as *Operations, Methods* and *Attributes*), *Packages* that can contain any kind of *ModelElement, Collaborations* that own a number of *ClassifierRoles* and *AssociationRoles*, etc.

Another important specialisation of *ModelElement* is *Relationship* (which has been added newly to the UML metamodel in version 1.3). It is used to represent relationships between elements. The *Relationship* metaclass has many different specialisations. For instance, a

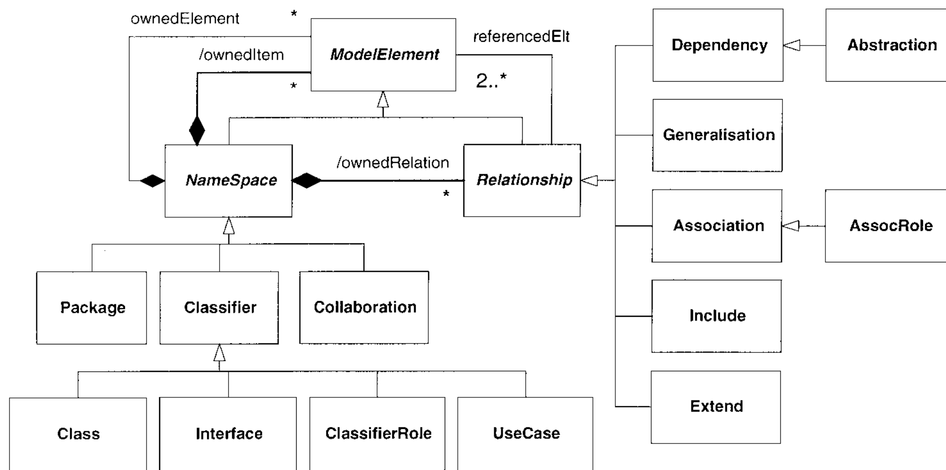

Figure 2. Distinguishing name spaces and relationships in UML.

Generalisation or *Association* relationship can be placed between *Classifiers*, an *Include* or *Extend* relationship can be put between *UseCases*, and a *Dependency* relationship can be used to connect any two (or more) *ModelElements*.

In figure 2, this basic structure is visualised. Some of the (derived) associations in the picture are not (yet) part of the UML metamodel and will be discussed in more detail later. Also note that we have enumerated only a representative selection of *NameSpaces* and *Relationships* for the sake of the presentation.

3.3. Evolution contracts in UML

The *Dependency* relationship is a general kind of relationship stating that the implementation or functioning of one or more *ModelElements* requires the presence of one or more other *ModelElements*. A *Dependency* relationship contains a *client* and *supplier* association role. The *client* requires the presence and knowledge of the *supplier* element, and a change to the *supplier* may affect the *client*.

An *EvolutionContract* can be defined by specialising the *Dependency* metaclass as depicted in figure 3. An *EvolutionContract* must be stereotyped to describe the specific kind of modification (i.e., the contract type) that takes place. The *modification* role of an *EvolutionContract* refers to a nonempty sequence of *ModelElements*, namely those *ModelElements* in the *client* or *supplier NameSpace* that will be modified, added or removed by the *EvolutionContract*. The exact semantics of the *modification* depends on the particular *stereotype* of the *EvolutionContract*, and is specified by extra well-formedness rules in OCL.

The approach explained above is similar to the *Abstraction* metaclass in the UML metamodel, which is a specialisation of *Dependency* that can be stereotyped to four different variants, namely *Derivation, Realization, Refinement* and *Trace* (with stereotypes

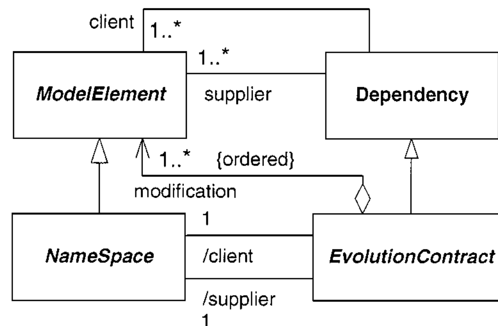

Figure 3. Evolution contract extension of the UML metamodel.

≪derive≫, ≪realize≫, ≪refine≫ and ≪trace≫, respectively). Unfortunately, *EvolutionContract* cannot be seen as a special kind of *Abstraction* because of the explicit constraint in (OMG, 1999) that "an abstraction is a relationship that relates two elements (or sets of elements) that represent the same concept at **different** levels of abstraction or from different viewpoints". Indeed, when dealing with evolution, we typically (but not exclusively) want to model relationships between concepts at the **same** level of abstraction.

There are some extra well-formedness constraints of *EvolutionContract* that can be specified independently of its *stereotype*. An *EvolutionContract* can only be defined between *NameSpaces* instead of arbitrary *ModelElements*. At least one modification needs to be made by the *EvolutionContract*. The (derived) *supplier* and *client* roles of an *EvolutionContract* each refer to exactly one *NameSpace* instead of a nonempty set of *ModelElements*. Additionally, the *supplier* and *client NameSpaces* must have the same type. In other words, we do not allow the type of an element to be changed during evolution. All of these restrictions are required by the underlying formalism defined in Mens (1999). However, in the future we might consider removing some of these restrictions, once we have formally investigated the effect of this on the detection of evolution conflicts. The following well-formedness rule formalises all of the above constraints:

```
context EvolutionContract
inv: self.supplier->size=1
inv: self.client->size=1
inv: self.modification->size>=1
inv: self.supplier.oclIsKindOf(NameSpace)
inv: self.supplier.oclIsTypeOf(self.client)
```

As mentioned before, the *stereotype* of an *EvolutionContract* corresponds to the contract type that describes how the *supplier* is modified. A number of primitive and composite contract types can be conceived that apply to every kind of UML model, thus yielding a general model for evolution. The primitive contract types make an explicit distinction between elements and relationships between elements, because it helps to formulate evolution conflict rules, and consequently allows us to detect more potential inconsistencies between parallel evolutions of the same model. In UML, this distinction between elements and relationships is made using the *NameSpace* and *Relationship* metaclasses. *NameSpace*

has a composite relationship *ownedElement* that refers to all the elements contained in the current element. Unfortunately, *ownedElement* refers to elements as well as relationships without making a distinction between them. Because this distinction is essential for our conflict detection mechanism, we need to define two additional OCL operations *ownedRelation* and *ownedItem* on *NameSpace*, to extract the relationships and non-relationships from *ownedElement*, respectively. In figure 2, these user-defined operations are shown as derived associations. Their exact OCL definition is given below:

```
context NameSpace::ownedItem():Set(ModelElement) inv:
  ownedItem = self.ownedElement->reject(oclIsKindOf(Relationship))
context NameSpace::ownedRelation():Set(Relationship) inv:
  ownedRelation = self.ownedElement->
  select(oclIsKindOf(Relationship))
```

In figure 2, we have also added an association *referencedElt* from *Relationship* to *ModelElement* to refer to all the elements that play a role in the relationship. Indeed, in the UML metamodel, there is currently no uniform or consistent way to refer to the elements in a relationship. A *Dependency* has a *client* and a *supplier*, a *Generalisation* has a *child* and a *parent*, an *Association* has a *connection*, etc. Therefore, all these association roles need to be related to the more general association role *referencedElt* as follows:

```
context Dependency inv:
  self.referencedElt = self.client->union(self.supplier)
context Generalisation inv:
  self.referencedElt = self.child->including(self.parent)
context Association inv:
  self.referencedElt = self.connection
```

In order to fully understand these constraints, one should consult the relevant parts of the UML metamodel. Also in the case of *NameSpaces* there are some inconsistencies in the use of role names. Therefore, the following well-formedness rules need to be introduced (no special constraints are needed for *Package* and *Collaboration*):

```
context Classifier inv:
  self.ownedItem = self.feature
context ClassifierRole inv:
  self.ownedItem = self.availableFeature
```

3.4. Evolution contract stereotypes

Using the above extensions to the UML metamodel, we can now take a closer look at the different kinds of *EvolutionContracts* based on their *stereotype*. We will make a distinction between primitive evolution contracts and composite ones. The stereotyped evolution contracts *PrimitiveEC* and *CompositeEC* are used as abstract metaclasses that are specialised into concrete subclasses. Indeed, because *Stereotypes* are defined as *GeneralizableElements* in the UML metamodel, we can directly create specialisations of *PrimitiveEC* and *CompositeEC*.

Table 1. Possible stereotypes for an *EvolutionContract*.

Stereotype	Name of stereotyped *EvolutionContract*	Meaning
	PrimitiveEC	*abstract stereotype that is specialised into one of the four stereotypes below*
≪add≫	Addition	Adding elements to a *NameSpace*
≪remove≫	Removal	Removing elements from a *NameSpace*
≪connect≫	Connection	Adding *Relationships* between the elements of a *NameSpace*
≪disconnect≫	Disconnection	Removing *Relationships* between the elements of a *NameSpace*
	CompositeEC	*abstract stereotype that is specialised into one of the two stereotypes below*
≪promotion≫	Promotion	Defining a high-level *EvolutionContract* as a set of lower-level ones
≪sequence≫	Sequentialisation	Defining an *EvolutionContract* as a sequence of smaller ones

PrimitiveEC corresponds to the most elementary modifications one can perform on a *NameSpace* and can be specialised into: adding elements to (*Addition*) or removing elements from (*Removal*) a *NameSpace*, and adding (*Connection*) or removing (*Disconnection*) relationships between the elements of the *NameSpace*.[1] *CompositeEC* denotes an evolution contract that is built up from simpler ones, and can be specialised into *Promotion* and *Sequentialisation*. All these stereotypes are summarised in Table 1, and are general enough to be applicable to any kind of UML model. Nevertheless, in some cases there is also a need for model-specific modifications. This can be done by defining user-defined specialisations of the basic *EvolutionContract* stereotypes, as will be illustrated in a later section.

In the next section we discuss which extra well-formedness constraints need to be added for primitive evolution contracts, and Section 5 explains how the contract types facilitate detection of evolution conflicts. Section 6 discusses some scalability issues, including composite contract types and user-defined specialisations.

4. Primitive evolution contracts

4.1. Addition

An *Addition* is used to add new *ModelElements* to a *NameSpace. Addition* can be used to add features to a class, to add classes to a namespace, to add use cases to a use case model, etc. An example was given already in figure 1, where a `Set` *Interface* is extended by adding new *Operations* `remove` and `intersection` to it. To the dependency arrow, which is adorned with stereotype ≪add≫, a constraint {`modification = [remove, intersection]`˝ is attached to specify the exact *modifications* that are being made.

The following well-formedness constraint can be defined for an *Addition*. Its *modification* must be a nonempty sequence of *ModelElements* that need to be added to the *supplier* of the corresponding *EvolutionContract*. *Relationships* cannot be added to the *supplier* by means of an *Addition*. Moreover, the added elements should not be present already in the *supplier*. The *client* is created by adding all these elements, while leaving the rest of the *supplier* (such as the relationships between existing elements) untouched.

```
context Addition
inv: self.modification->forAll(not oclIsKindOf(Relationship))
inv: self.supplier.ownedRelation = self.client.ownedRelation
inv: self.supplier.ownedItem->intersection(self.modification)->
     isEmpty
inv: self.supplier.ownedItem->
     union(self.modification)=self.client.ownedItem
```

The reason why *modification* in figure 3 refers to a sequence of *ModelElements* rather than a sequence of *NameSpaces* (as might be expected) is illustrated in the example of figure 1, where the *modification* of the *EvolutionContract* is a sequence of *Operations*, while *Operation* is not a specialisation of *NameSpace*.

Because of the constraint on *EvolutionContract* that *supplier* and *client* must have the same type, and because of the well-formedness constraints associated with this *supplier* and *client*, it follows that the type of *ModelElement* that can be added to a certain *supplier* by means of an *Addition* depends on the type of the *supplier*. For example, in figure 1 the type of the *supplier* is *Interface*, which means that an *Addition* is only allowed to add elements of type *Operation*, because otherwise the well-formedness constraints of the *client* (that must also be of type *Interface*) would be breached.

4.2. Connection

A *Connection* is used to add *Relationships* between existing *ModelElements* in a *NameSpace*. It can be used to add *Associations* or *Generalisations* between classes, to add *Include* or *Extend* relationships between use cases, to add message invocations between objects, etc. An example is given in figure 4, where a model containing classes `Set` and `Element` is refined by adding an association with roles `owner` and `owned` between these classes. Obviously, a *Connection* can also be used to add multiple relationships at once.

The following well-formedness constraints can be defined for a *Connection*. Its *modification* is a nonempty sequence of *Relationships* that all need to be added to the *supplier*. The *client* is then created by adding these *Relationships*, and leaving the rest untouched. Moreover, new *Relationships* can only be added between *ModelElements* that already exist in the *supplier*.

Because OCL does not explicitly define an operation to determine whether a certain set is a subset of another one, we have defined it ourselves using the mathematical property that $A \subseteq B \Leftrightarrow A = A \cap B$.

```
context Set::subset(set2:Set(T)):Boolean inv:
  self = self->intersection(set2)
```

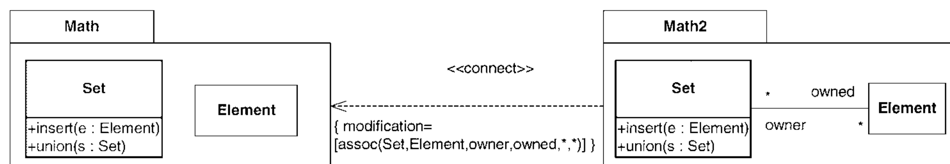

Figure 4. Applying evolution contracts to evolution of class diagrams.

Using this definition, we can give the OCL constraints for *Connection*:

```
context Connection
inv: self.modification.oclIsKindOf(Sequence(Relationship))
inv: self.modification->forAll(referencedElt->
     subset(self.supplier.ownedItem))
inv: self.supplier.ownedItem = self.client.ownedItem
inv: self.supplier.ownedRelation->
     intersection(self.modification)->isEmpty
inv: self.supplier.ownedRelation->
     union(self.modification)=self.client.ownedRelation
```

As with *Addition*, the types of *Relationships* that can be added to a certain *supplier* by means of a *Connection* depend on the type of the *supplier*.

4.3. Removal and disconnection

The third kind of primitive evolution contract, a *Removal*, is used to remove existing *ModelElements* from a *NameSpace*. It can be seen as the inverse of *Addition*. The well-formedness constraints for *Removal* are the same as for *Addition*, except that the roles of *supplier* and *client* have been inverted. Moreover, an extra condition is needed to ensure that there can be no dangling references after *Removal*.

```
context Removal
inv: self.modification->forAll(not oclIsKindOf(Relationship))
inv: self.supplier.ownedRelation = self.client.ownedRelation
inv: self.client.ownedItem->
     intersection(self.modification)->isEmpty
inv: self.client.ownedItem->
     union(self.modification)=self.supplier.ownedItem
inv: self.supplier.ownedRelation->forAll(
       referencedElt->intersection(self.modification)->isEmpty)
```

A similar reasoning can be made for *Disconnection*, which is used to remove existing *Relationships* between *ModelElements* in a *NameSpace*. It can be seen as the inverse of *Connection*. An example is given horizontally in figure 5, where an operation invocation from `union` to `insert` is removed from the specialisation interface of a class. More details will be given in the next section.

```
context Disconnection
inv: self.modification.oclIsKindOf(Sequence(Relationship))
inv: self.modification->forAll(referencedElt->
     subset(self.supplier.ownedItem))
inv: self.supplier.ownedItem = self.client.ownedItem
inv: self.client.ownedRelation->
     intersection(self.modification)->isEmpty
inv: self.client.ownedRelation->
     union(self.modification)=self.supplier.ownedRelation
```

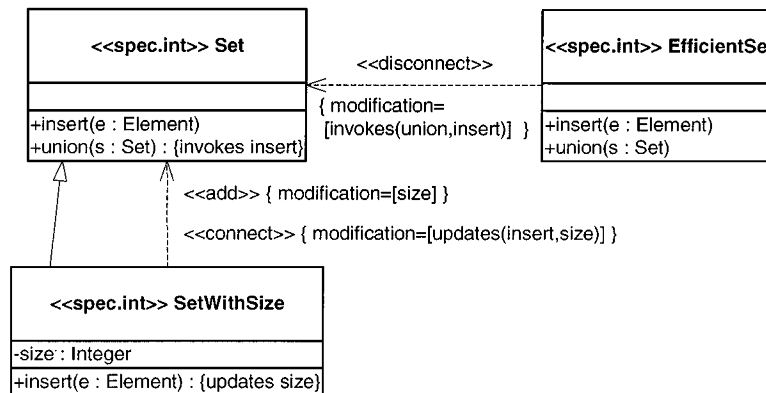

Figure 5. Evolution of class inheritance hierarchies.

5. Detecting evolution conflicts

This section describes how evolution contracts aid in detecting evolution conflicts. For a more detailed discussion of all the different aspects involved in conflict detection we refer to Lucas (1997) and Mens (1999).

5.1. Example

Evolution contracts aim at maintaining a maximum degree of consistency between evolving model elements and the models in which they are used. More specifically, evolution contracts provide feedback on possible *upgrade conflicts* that occur due to the evolution of the model elements. Evolution problems can also occur when two parallel modifications are made to the same model element. A *merge conflict* arises when the combination of independent modifications leads to undesired interactions.

Figure 5 illustrates an upgrade conflict. Instead of using class interfaces, we make use of specialisation interfaces (Lamping, 1993) which also specify the relevant invocation dependencies between operations. For example, the (specialisation) interface `Set` contains an operation `union` that invokes `insert`, the second operation of `Set`. `Set` is specialised to `SetWithSize` which overrides the `insert` operation so that it additionally accesses and updates the `size` attribute each time a new element is added to the set. This specialisation is achieved in UML by putting a *Generalization* relationship between `SetWithSize` and `Set`. Because this *Generalization* does not explicitly document the exact modifications that have been made, while this information is essential to enable conflict detection, we need to add an *EvolutionContract* dependency between `Set` and `SetWithSize` as well. This dependency specifies that `SetWithSize` is incrementally obtained from `Set` by adding the attribute `size` (*Addition*) and accessing the attribute `size` from the operation `insert` (*Connection*). In fact, the *EvolutionContract* between `Set` and `SetWithSize` is a composite evolution contract because it is composed of two

more primitive evolution contracts *Addition* and *Connection*. This will be discussed in more detail in Section 6.

In the horizontal direction, `Set` evolves into a new version `EfficientSet` by removing the invocation of `insert` by `union` for efficiency reasons. This is expressed by means of an *EvolutionContract* dependency between `Set` and `EfficientSet`. It is a *Disconnection*, since it removes an invocation relationship between two operations.

To find out if upgrading `Set` to `EfficientSet` leads to unexpected results, we need to know whether `SetWithSize` is still a valid and meaningful specialisation of `EfficientSet`. Unfortunately this is not the case, since originally the invocation of `union` in `SetWithSize` leads to an indirect update of `size`, while this is not the case anymore if we substitute `Set` by `EfficientSet`. As a result, performing a union of sets in the upgraded version will not cause the size of the set to increase, while it obviously should.

This problem is called an "inconsistent operation conflict". It does not only arise when upgrading parts of a UML model to a new version, but also when merging parallel modifications that have been made by different software developers to the same model. For example, when we start from a model containing only the `Set` interface, one developer could decide to change this `Set` into an `EfficientSet`, while another developer could decide to add a new interface `SetWithSize` as a specialisation of `Set`. When merging both modifications together, we obtain exactly the same conflict as before.

5.2. *Conflict detection*

In general, evolution conflicts can be detected by comparing different *EvolutionContracts* with the same *supplier*. To automate conflict checking, the contract types can be used to set up conflict tables that describe what kinds of conflicts may occur in which cases. For example, the "inconsistent operation" explained above occurs every time that one modification is a *Disconnection* of a certain operation invocation and the other modification is a *Connection* from the operation that was invoked.

When detecting evolution conflicts, a distinction should be made between *structural conflicts* and *behavioural conflicts*. Structural conflicts correspond to structural problems, and give rise to an inconsistent model when both modifications are merged. An example of such a situation is a dangling reference. It arises when a *Connection* and a *Removal* are merged, where the *Removal* deletes one of the elements used by the *Connection*. Behavioural conflicts are more subtle, since they only give rise to a behavioural problem. The two modifications do not interact in the way they are supposed to. An example of this is the inconsistent operation conflict of the previous section. Since it is very difficult to know how an interaction is supposed to behave, evolution contracts assume a worst case scenario, and only detect potential sources of problems. As such, the detected evolution conflicts should be considered merely as *warnings*, because they indicate situations where something might have gone wrong. To find out whether these warnings correspond to actual incompatibilities, more behavioural information is needed. To this extent, one could rely on other formal techniques for performing deadlock detection, data and control flow analysis, etc. For a more detailed discussion on all different aspects involved in conflict detection, we refer to Mens (1999).

6. Scalability

The four primitive evolution contracts discussed in Section 4 are the basis for a general approach to evolution of UML models, but they are too elementary to cope with more complex situations.

- Evolution contracts must be applicable to model elements of any size. One could apply evolution contracts to model elements as small as classes, or as complex as entire software models. A general way to group arbitrary model elements together is by making use of *Packages*.
- When model elements can be nested by means of a package mechanism, the need arises to look at evolution contracts at different levels of abstraction, and to promote (respectively, demote) contracts to a higher (respectively, lower) level of abstraction.
- Besides primitive evolution contracts, arbitrarily complex combinations of evolution contracts are needed. Since certain combinations will occur more frequently than others, we need to provide the possibility to introduce certain composite evolution contracts as predefined combinations of primitive evolution contracts.
- While the basic evolution contracts are sufficient in most situations, for some special kinds of model elements new specific contract types will be needed. These can be specified very easily by adding a user-defined stereotype to the *EvolutionContract*.

In the following subsections we will discuss each of the above issues in more detail.

6.1. Nested model elements

The UML *Package* mechanism can be used to nest *ModelElements*, which can be *Packages* again. In this way arbitrarily complex components can be created. Moreover, different kinds of *ModelElements* can be combined together in this way. As an example, a design component could be defined as a stereotyped ≪design≫ package which owns a ≪collaboration≫ subpackage containing a *Collaboration*, and an ≪interaction≫ subpackage containing an *Interaction*. Since the UML semantics specifies that an *Interaction* should always have a *Collaboration* as *context*, an explicit *Dependency* is put from the ≪interaction≫ package to the ≪collaboration≫ package. A more detailed discussion about how to deal with evolution of this particular kind of components was presented in Mens et al. (1999).

The use of packages does not require any changes to the evolution contract definition, since *Package* is defined as a specialisation of *NameSpace* in the UML metamodel, as shown in figure 2.

6.2. Composite evolution contracts

Not only the model elements themselves can be arbitrarily complex, but also the way in which these model elements can evolve. To this extent, the notion of a composite evolution contract needs to be introduced. Composite evolution contracts are evolution contracts that are composed out of several other ones. This is expressed by the following OCL constraint.

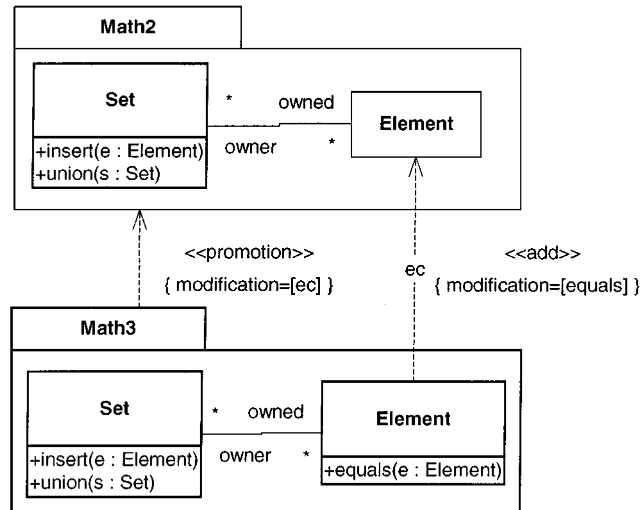

Figure 6. Promoted evolution contract.

```
context CompositeEC inv:
  self.modification.oclIsKindOf(Sequence(EvolutionContract))
```

We will now look in more detail at two useful specialisations of *CompositeEC*: *Promotion and Sequentialisation*.

When *Packages* are used to nest *ModelElements* in each other, it is necessary to look at *EvolutionContracts* at different levels of abstraction. We therefore introduce the possibility to automatically promote any number of low-level *EvolutionContracts* to an *EvolutionContract* between the higher-level elements. A example of such a *Promotion* evolution contract is given in figure 6, where a package `Math2` evolves into a new package `Math3` by extending the class `Element` with an extra operation `equals`. This low-level *Addition* (to which we have attached the name `ec`) between classes leads to a *Promotion* between the packages in which they are nested. In general, any sequence of lower-level evolution contracts may be composed into a single *Promotion*.

The extra constraint on the *modification* role of a *Promotion* is that each *EvolutionContract* in the sequence must be defined between elements that are owned by the *supplier* and *client* of the *Promotion* respectively. This constraint allows us to cross only one level of abstraction, but in a similar way a more sophisticated kind of promotion could be defined to go up an arbitrary number of levels.

```
context Promotion inv:
  self.modification->forAll(ec |
    self.supplier.ownedItem->includes(ec.supplier) and
    self.client.ownedItem->includes(ec.client) )
```

As certain sequences of primitive evolution contracts are used more frequently than others, we provide the possibility to define a *Sequentialisation* evolution contract as a sequence

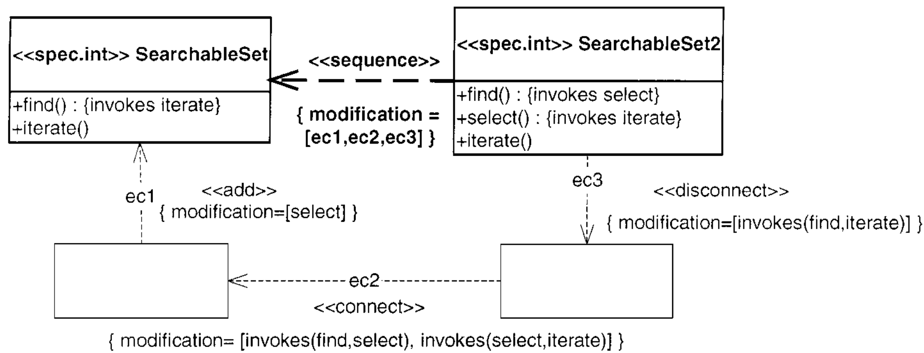

Figure 7. Sequentialisation evolution contract.

of other ones. An example of such a *Sequentialisation* is presented in figure 7, which shows a factorisation of operation invocations in a `SearchableSet` class: a direct invocation of operation `iterate` by operation `find` is replaced by an invocation through an extra indirection, by introducing an auxiliary operation `select`. This composite *EvolutionContract* is defined as a sequence of three more primitive evolution contracts *Addition, Connection* and *Disconnection*. The top part of figure 7 shows the *Sequentialisation* evolution contract, the bottom part shows the sequence of primitive evolution contracts (named `ec1`, `ec2` and `ec3`) it is composed from.

The *client* of a *Sequentialisation* is obtained by successively applying each *Evolution-Contract* in the sequence. This leads to the following OCL constraint:

```
context Sequentialisation
inv: self.supplier = self.modification->first.supplier
inv: self.client = self.modification->last.client
inv: Sequence{1..(self.modification->size-1)}->forAll( n |
       self.modification->at(n).client =
       self.modification->at(n+1).supplier )
```

6.3. *Predefined composite evolution contracts*

Although the notions of *Promotion* and *Sequentialisation* can be used to construct complex evolution contracts out of simpler ones, it is not enough from a practical point of view. We also need a mechanism to capture often recurring patterns of evolution contracts. For example, the evolution contract of figure 7 arises in many different situations. Essentially, it factors out some part of the behaviour of one or more elements into an intermediary element. It can be considered as a *Sequentialisation* evolution contract to which some extra constraints are attached. More specifically, it always consists of an *Addition, Connection* and *Disconnection*, with the additional condition that the *Connection* can only add new relationships of which the source element was introduced by the *Addition*, and a similar

constraint for the *Disconnection*. The details can be found in Lucas (1997). To formally capture this pattern, we define *Factorisation* as a specialisation of *Sequentialisation*, and define the following constraints for it:

```
context Factorisation inv:
  let ext = self.modification->select(oclIsTypeOf(Addition)) in
  let con = self.modification->select(oclIsTypeOf(Connection)) in
  let dis = self.modification->select(oclIsTypeOf(Disconnection)) in
  self.modification = ext->union(con)->union(dis) and
  con->intersection(dis)->isEmpty and
  dis->forAll(referencedElt->subset(self.supplier.ownedItem) ) and
  con->forAll(referencedElt->subset(self.supplier.ownedItem->
       union(ext)))
```

Obviously, all constraints of *Sequentialisation* are automatically inherited by *Factorisation* as well. To be complete, some other constraints should be added to *Factorisation*, but these become very complex and would only clutter the example.

Needless to say, the same technique can also be used to create user-defined specialisations of *Promotion*. For example, we could define a *PromotedAddition* as a specialisation of *Promotion* with the additional constraint that it is built-up from lower-level *Additions*.

```
context PromotedAddition inv:
  self.modification.oclIsKindOf(Sequence(Addition))
```

6.4. *Model-specific evolution contracts*

For some specific software artifacts, the basic evolution contracts are insufficient to express the exact modification that occurs. In those cases, we need to define model-specific specialisations of existing *EvolutionContracts*. For example, suppose that we want to model the evolution of an abstract class to a concrete one. By looking more closely to the UML metamodel we can generalise this to any kind of *GeneralizableElement*, that can be made concrete by changing the value of the attribute `isAbstract` from `true` to `false`. To formalise this evolution step, we need to define a *Concretisation* as a user-defined specialisation of *PrimitiveEC*, with the additional constraint that it can only be applied to a *GeneralizableElement* (or any subclass thereof). The exact OCL constraints are given below.

```
context Concretisation
inv: self.supplier.oclIsKindOf(GeneralizableElement)
inv: self.supplier.isAbstract = true
inv: self.client.isAbstract = false
```

6.5. *Scaling up conflict detection*

The different ways of scaling up evolution contracts mentioned in the previous subsections are necessary to apply evolution contracts to larger models. The use of *Packages* and

composite contract types gives us a flexible way to look at models at the desired level of abstraction. The possibility to create new user-defined (primitive and composite) evolution contracts allows us to customise the formalism to particular situations.

Crucial to the scalability is how detection of evolution conflicts behaves in presence of composite evolution contracts. In the case of *Promotion* evolution contracts, all conflicts detected at a lower level will be "promoted" to the higher level. When *Sequentialisation* evolution contracts are involved, things are more complicated. As a first approximation, conflicts caused by a *Sequentialisation* evolution contract can be detected by considering the *Primitive* evolution contracts from which it is made up. This can however lead to the detection of too many potential conflicts, as subsequent subcontracts in the *Sequentialisation* may cause local conflicts to be annihilated. For example, if an *Addition* with a certain element is followed by a *Removal* of the same element, the conflicts caused by the *Addition* should not be considered. Therefore, before detecting evolution conflicts, the sequence should first be transformed in such a way that each evolution contract is independent—with respect to the possible conflicts—of the preceding ones. This process of removing redundancy in an arbitrary evolution sequence is called "normalisation" and is treated in detail in Mens (1999).

More importantly, the use of predefined high-level evolution contracts allows the software developer to give extra feedback on which conflicts are real and which probably are not. A good example can be found with *Factorisation*. An "inconsistent operation" conflict (as explained in figure 5) occurs when an invocation of an operation is removed (*Disconnection*) by one software developer and this same operation is independently modified by another developer to invoke another operation (*Connection*). Since a *Factorisation* involves a *Disconnection*, an inconsistent operations conflict can appear after a *Factorisation*. Upon closer examination however, we see that there is an intuitive difference between a stand-alone *Disconnection* and a *Disconnection* that is part of a *Factorisation*. When an operation invocation is removed during a *Factorisation*, this invocation 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, by explicitly declaring the *Disconnection* to be part of the *Factorisation*, the user indicates that the inconsistent operation conflict can be ignored.

7. Tool support

It is obvious that the mechanism of composite evolution contracts and conflict detection is difficult to manage without proper tool support. Therefore, the formalism should be incorporated into a CASE tool. The development of the evolution contract methodology and its incorporation in tools have always been strongly related. On the one hand, the incorporation of evolution contracts in tools and the use of these tools in case studies has been invaluable for the further development of the evolution contract formalism. On the other hand, the existence of a formal specification of evolution contracts facilitates its incorporation into tools.

As evolution contracts were originally conceived for implementation-level components (Steyaert et al., 1996), the first tools aimed at supporting evolution at the implementation level. Tools were developed in Smalltalk to semi-automatically extract evolution contracts

from existing Smalltalk code. In his PhD dissertation, Koen De Hondt further elaborated upon this idea, which led to a reverse engineering tool for Smalltalk (De Hondt, 1998). Among others it is capable of semi-automatically extracting class collaborations from Smalltalk code, and representing these collaborations graphically in a CASE tool like Rational Rose™.

Closer to the topic of this paper are tools that provide support for evolution at analysis and design level. We have already carried out some interesting experiments in UML CASE tools like Rational Rose™, Select Enterprise™ and Visio Professional™, in which we added limited support for evolution contracts by means of the built-in scripting languages.

For the purpose of merging parallel evolutions of the same UML model, the *Visual Differencing* tool (which comes with Rational Rose™) can be used. Unlike the evolution contracts approach, this tool is only capable of detecting and resolving *structural* inconsistencies (i.e., ill-formed UML models). A similar remark can be made for the merge tool proposed in Westfechtel (1991), with the distinction that this latter approach is language-independent. The essence of evolution contracts is that they enable detection of more *behavioural* inconsistencies. In practice, these behavioural merge conflicts are the more important, since they are much harder to detect manually and give rise to more subtle problems.

We have implemented a declarative evolution contract framework in PROLOG, in such a way that it can be customised easily to many different domains. This makes it possible to check evolution conflicts in evolving UML models by exporting them to PROLOG and performing the conflict detection there. This experiment is part of a large case study (containing more than 600 design classes) that is currently being carried out with an industrial partner. The main goal is to illustrate the practical benefits of evolution contracts, and to find out where the formalism still needs to be enhanced. Especially the scalability aspects will be investigated further.

To make the tool more user-friendly, we still need to experience in practice which user-defined contract types could be defined. In this paper we already briefly discussed some, such as *Factorisation* and *Concretisation*, but there are many more useful contract types that can be imagined. Some of these user-defined contract types (like *Factorisation*) will be domain-independent, while others (like *Concretisation*) will only be applicable in very specific cases. In Lucas (1997) and Mens (1999), a number of user-defined contract types has been defined and discussed. Refactoring and restructuring operations (Opdyke, 1992) are also interesting candidates for composite contract types.

Finally, a tool can be very helpful in assisting conflict resolution once evolution conflicts have been detected. A first attempt towards such a tool has been undertaken in the context of evolution of Smalltalk code (Mezini, 1997). This approach could be generalised and extended to allow semi-automatic conflict resolution for UML models as well.

8. Summary and future work

This paper showed how the evolution contract formalism can be integrated into the UML metamodel in a straightforward way. As a result, we obtain a general mechanism for dealing with unanticipated evolution of arbitrary UML models. We characterised a number of basic evolution contracts that can be uniformly applied to many different UML models. Scalability

mechanisms were provided to deal with evolution at different levels of abstraction, and to allow model-specific or domain-specific modifications to be defined as well.

By formally documenting model evolution with evolution contracts, incompatibilities or undesired behaviour can be detected when part of a model is upgraded to a new version, or when different software developers independently make changes to the same or related parts of a model. Because of our uniform approach, conflict detection becomes independent of the specific kind of model that is under consideration. Another important benefit of our approach is that its formality makes it easy to integrate in CASE tools. A number of prototype tools have already been developed and are currently being validated in industrial case studies.

A very interesting application of the ideas in this paper would be to employ evolution contracts for managing evolution of the UML metamodel itself. Each time a new version of the UML metamodel (currently 1.3) is delivered, it is possible that UML models that have been developed in the previous version become invalidated. Provided that we document the evolution of the UML metamodel with evolution contracts, it becomes possible to automatically identify the potential problem areas for each particular case.

An important topic that has not yet been addressed is the application of evolution contracts to monitor evolution between *different* kinds of model elements. Until now, evolution contracts were restricted to have the same type of supplier and client. It could also be useful to apply evolution contracts to document the relationship between different kinds of model elements ranging from requirements to implementation phase. Possible evolutions could be: a use case which is refined into a collaboration (*Refinement*), an interface which is transformed into a class (*Realization*), etc. In order for *Refinements* and *Realizations* (two stereotyped *Abstractions*) to benefit from the techniques of evolution contracts, the UML metamodel should be altered so that *Abstraction* becomes a specialisation of *Evolution-Contract*. Additionally, some extra changes will need to made to the evolution contract formalism in order to be still valid in this more general context.

Acknowledgments

We express our gratitude to our industry partners at MediaGeniX and Getronics for giving us the necessary practical feedback. We thank Claudia Pons, Tom Tourwe, Bart Wouters, Maja D'Hondt and especially Veronica Argañaraz, Carine Lucas and Kim Mens for proof-reading our paper and providing very valuable comments. We also thank all the anonymous referees for the careful review of this paper and the many suggestions for improvements they provided.

Note

1. In previous work on reuse contracts, the terms *Refinement* and *Coarsening* were consistently used instead of *Connection* and *Disconnection*. Unfortunately, the term *Refinement* is reserved in UML to denote a stereotyped *Abstraction* (which is a special kind of *Dependency*). Similarly, *Extension* and *Cancellation* were used instead of *Addition* and *Removal*, but the metaclass *Extend* is already used in UML to denote a *Relationship* between *UseCases*.

References

Bohner, S.A. and Arnold, R.S. 1996. *Software Change Impact Analysis*. IEEE Press.

De Hondt, K. 1998. A novel approach to architectural recovery in evolving object-oriented systems. Ph.D. Dissertation, Department of Computer Science, Vrije Universiteit Brussel, Belgium.

D'Hondt, M. 1998. Managing evolution of changing software requirements. Dissertation, Department of Computer Science, Vrije Universiteit Brussel, Belgium.

Ecklund, E.F., Jr. Delcambre, L.M.L., and Freiling, M.J. 1996. Change cases: Use cases that identify future requirements. In *Proc. OOPSLA '96. ACM SIGPLAN Notices*, 31(10):342–358.

Evans, A., France, R., Lano, K., and Rumpe, B. 1999. The UML as a formal modelling notation. In *Selected Papers of ≪UML≫'98 International Workshop*. Lecture Notes in Computer Science, Vol. 1618, Springer-Verlag, pp. 336–348.

Gogolla, M. and Richters, M. 1998. On constraints and queries in UML. *The Unified Modeling Language—Technical Aspects and Applications*. Physica-Verlag.

Hamie, A., Howse, J., Kent, S., Mitchell, R., and Civello, F. 1999. Reflections on the object constraint language. In *Selected Papers of ≪UML≫'98 International Workshop*. Lecture Notes in Computer Science, Vol. 1618, Springer-Verlag, pp. 162–172.

Helm, R., Holland, I.M., and Gangopadhyay, D. 1990. Contracts: specifying behavioral compositions in object-oriented systems. In *Proc. OOPSLA/ECOOP'90. ACM SIGPLAN Notices*, 25(10):169–180.

Jacobson, I., Griss, M., Jonsson, P. 1997. Making the reuse business work. *IEEE Computer*.

Kent, S., Evans, A., and Rumpe, B. 1999. UML semantics FAQ. In *ECOOP '99 Workshop Reader*. Lecture Notes in Computer Science, Springer-Verlag.

Kiczales, G. and Lamping. J. 1992. Issues in the design and documentation of class libraries. In *Proc. OOPSLA'92. ACM SIGPLAN Notices,* 27(10):435–451.

Lamping, J. 1993. Typing the specialisation interface. In *Proc. OOPSLA'93. ACM SIGPLAN Notices*, 28(10):201–214.

Lucas, C. 1997. Documenting reuse and evolution with reuse contracts. Ph.D. Dissertation, Department of Computer Science, Vrije Universiteit Brussel, Belgium.

Mens, T. 1999. A formal foundation for object-oriented software evolution. Ph.D. Dissertation, Department of Computer Science, Vrije Universiteit Brussel, Belgium.

Mens, T., Lucas, C., and Steyaert, P. 1999. Supporting reuse and evolution of UML models. In *Selected Papers of ≪UML≫'98 International Workshop*. Lecture Notes in Computer Science, Vol. 1618, Springer-Verlag, pp. 378–392.

Mezini, M. 1997. Maintaining the consistency of class libraries during their evolution. In *Proc. OOPSLA'97. ACM SIGPLAN Notices*, 32(10):1–21.

Object Management Group. 1999. Unified modeling language specification version 1.3. OMG Document ad/99-06-08.

Opdyke, W.F. 1992. Refactoring object-oriented frameworks. Ph.D. Dissertation, University of Illinois at Urbana-Champaign. Technical Report UIUC-DCS-R-92-1759.

Övergaard, G. and Palmkvist, K. 1999. A formal approach to use cases and their relationships. In *Selected Papers of ≪UML≫'98 International Workshop*. Lecture Notes in Computer Science, Vol. 1618, Springer-Verlag, pp. 406–418.

Richters, M. and Gogolla, M. 1998. On formalizing the UML object constraint language OCL. In *Proc. Int. Conf. Conceptual Modeling*, Springer-Verlag.

Romero, M. 1999. Managing architectural evolution with reuse contracts. Masters Dissertation, Department of Computer Science, Vrije Universiteit Brussel, Belgium.

Rubin, K.S. and Goldberg, A. 1992. Object behaviour analysis. *Communications of the ACM* (Special Issue on Object-Oriented Methodologies), 35(9):48–62.

Steyaert, P., Lucas, C., Mens, K., and D'Hondt, T. 1996. Reuse contracts: Managing the evolution of reusable assets. In *Proc. OOPSLA '96. ACM SIGPLAN Notices*, 31(10):268–286.

Westfechtel, B. 1991. Structure-oriented merging of revisions of software documents. In *Proc. 3rd Int. Workshop on Software Configuration Management*, ACM Press, pp. 68–79.