A Formal Approach to Relationships in the Unified Modeling Language

Gunnar Övergaard

Department of Teleinformatics
Royal Institute of Technology
Stockholm, Sweden
gunnaro@it.kth.se

Abstract

We present parts of a formal specification of the Unified Modeling Language. The paper focuses on the relationship constructs in UML, such as Association, Import and different kinds of Generalization.

Keywords: Formal specification, Object-oriented modelling, Relationship, Unified Modeling Language

1 Introduction

The *Unified Modeling Language* (UML) is a modelling language for modelling object-oriented systems. The language is used for specification, visualization and construction of the artifacts of object-oriented systems [UML97]. UML originates from the Booch method [Boo91], OOSE [JCJÖ93] and OMT [RBP+91], but has also been influenced by other sources, like ROOM [SGW94], OOram [RWL96] and Statecharts [HN96, HG97]. UML has recently become standard within The Object Management Group (OMG) for object-oriented analysis and design [LO96].

The specification of the language [USE97] is "semi-formal", i.e. parts of it are specified with well-defined languages while other parts have been described informally in English. The abstract syntax of the different language constructs in UML is specified with the graphical notation of Class diagrams in UML itself, while the well-formedness rules of UML are given in OCL, an object-oriented constraint language (see [UML97] for more details). Ordinary English is chosen for describing the semantics of UML. This makes the structure of the language rigorous whereas the semantics of the language is still quite informal.

It is commonly accepted that a language needs a formal specification to be unambiguous. Furthermore, the semantics of the language must be precise if tools are to perform intelligent operations on models expressed in the language, like consistency checks and transformations from one model to another. During the development of UML 1.1 the UML 1.1 task force, in which the author participated, had to find a balance between formalism and readability to avoid making the language definition overly complex for the current needs. However, we are aware that a more formal specification of UML will probably be needed in the future [USE97, 10].

This paper presents some of the results we have achieved in a project formalizing UML. It focuses on some of the major constructs for expressing relationships in UML and gives detailed specifications of these constructs. Other attempts to give UML a formal definition (see e.g. [BHK+97, KR97, SF97]) do not give a complete or in-depth definition of the language. Thus, many of the aspects of the relationship constructs have often been simplified or omitted.

We have chosen to use an operational semantics when formalizing UML. Since the intended user will be familiar with the object-oriented paradigm, the use of an object-oriented specification method instead of a traditional one, will cause no conceptual shift between the specification language and UML.

The paper is organized as follows: the next section includes a brief introduction to UML. In section 3 follows a presentation of the specification technique we have used. Sections 4-7 consist of a formal specification of important relationship constructs in UML. The paper ends with some concluding remarks.

2 The Unified Modeling Language

In this section we will give a brief introduction to UML. For a more comprehensive description we refer the reader to the language definition [USE97]. UML is a language for modelling and documenting the design of object-oriented systems. The language consists of a set of constructs common to most object-oriented modelling languages as well as a set of constructs useful for modelling large systems. All object-oriented languages contain at least one construct for declaration of the features of the different instances. A typical example of such a construct is *class* but other kinds also exist. In UML these constructs are called Classifiers. A *Classifier* declares a set of structural and behavioural features, such as Attributes and Operations. Each feature has a *visibility* property in the Classifier which indicates if the feature is visible and may be used outside the Classifier.

An *Association* defines a semantic connection between Classifiers. Instances of the Classifiers are connected with *Links* to each other corresponding to the Associations between the Classifiers. Furthermore, an Association may have features of its own, like Attributes and Operations, which are used for expressing information that is related to the relationship itself rather than to its connected Classifiers. In this case the Association is called an *Association class.*

Another type of connection between Classifiers is the *Generalization* relationship. This relationship means that all features of the more general Classifier are also present in the more specialized Classifier. Each Classifier has two features stating whether the Classifier may have Generalizations to other Classifiers, and if there may be Generalizations referring to the Classifier.

Classifiers are typically used for creating *Instances.* An Instance has a set of *Attribute links* holding the actual values of its Attributes, and it is connected to a set of Links. These two sets match the sets of Attributes and Associations of the Instance's Classifier, respectively. Each Operation which is declared in a Classifier may be applied to an Instance originating from that Classifier. An *Action* is a specification of an "executable" statement.

UML defines a collection of different kinds of Classifiers. A typical example of a Classifier is the *Class* construct, whose Instances are called *Objects.* Other kinds of Classifiers in UML are:

Data type — a declaration of a type whose Instances are primitive values

Interface — a collection of Operations

Use case — a specification of a set of sequences of Actions which an element can perform when it interacts with a user

Subsystem — a collection of Classifiers and other elements together with an abstract specification of their collective behaviour

In UML, *Package* is a construct which is used for grouping elements together. A Package can contain other Packages, Classifiers, and Associations and it may have Generalizations to other Packages. However, a Package cannot be instantiated and its only semantics is that it defines a namespace, i.e. each contained element must have a unique name within the namespace. An *Import* relationship between Packages implies that the elements in the imported Package are available in the importing Package, possibly under a new name. However, each element is still owned by only one Package, namely the Package in which the element is defined. The importing Package has no ownership of the referenced elements. When a Package is removed, all its owned elements are also removed, but not the referenced elements. The contents of a Package is the union of both the owned and the referenced elements. Subsystem is a special kind of Package which has an additional specification part describing what behaviour can be performed by the contained elements.

UML also includes Collaborations, Interactions and State machines. A *Collaboration* specifies how Instances of the participating Classifiers collaborate to perform a specific task, e.g. how they jointly perform the sequence of Actions specified by a Use case. An *Interaction* specifies the details of the cooperation between the Instances, e.g. what Messages are sent between the Instances and what their internal order is. A *State machine* is a specification of the sequence of *States* an Instance goes through during its life time in response to Events, such as reception of a Signal, or expiration of a time period. A State machine also includes the Instance's reaction to these Events.

There are also several different kinds of diagrams defined in UML which are to be used when documenting a model, e.g. Class diagram, Collaboration diagram and Activity diagram.

In the following sections we will present the semantics of some of the major relationship constructs. In section 4 the relationship between a namespace and its owned elements is presented. The Import relationship is presented in section 5 together with the relationship between a namespace and its referenced elements. The Association construct is presented in section 6, and some different semantics of the Generalization relationship are covered in section 7.

3 Specification Technique

Before discussing the semantics of the different relationship constructs in UML we will introduce the technique used for specification of the semantics of these constructs. We give an operational semantics of UML using an object-oriented specification language named ODAL which has been formalized using the π-calculus [MPW92]. ODAL is a simple, strongly typed language with a familiar syntax. It is used as the specification language in a framework for formal specification of modelling languages. (More information about ODAL is found in the appendix.) In this paper we use a simplified version in which e.g. all type information has been omitted. We have also omitted the description of the underlying mechanism for invocation of the operations.

In our specification of UML we use a technique with *maps* between the specification objects as primitive *specification* constructs. Each map has an *origin* (a specification object in the domain of the map) and a *value* (a set of specification objects in the codomain of the map). In principle, a *specification object* specifies the semantics of a non-relationship construct. The specification object contains, apart from its specific features, also a collection of maps – the subset of all maps in which the specification object is the origin. Typically, constructs in UML, like Class and Object, are specified with a specification object, while relationships, like Import and Generalization, are specified with maps. Sometimes a combination of both specification objects and maps is needed for the specification of a construct, like Association, due to the more complicated nature of those constructs.

A fundamental principle in our specification is that the semantics of a construct should be localized and not implicitly included in the specifications of other constructs. The reason for this is not only the need for readability and understandability but also for changeability of the specification. UML includes e.g. the Stereotype construct and when it is applied to an instance of a construct, it may modify the semantics of that instance. It would be much more difficult to understand the implication of applying a particular stereotype and to keep the specification consistent if the specification of the construct was not localized. This localization principle implies a specification technique of UML in which the semantics of the different kinds of relationships are separated from other kinds of constructs.

For example, the specification of an object construct states that an object has, among other things, a set of relationships, but the specification does not include any of their semantics. The meaning of a particular kind of relationship is defined separately. Hence, when a new kind of object construct is defined in the modelling language, features like atomic transactions, persistence etc. are included in the definition, but not features based on a relationship of a specific kind. Similarly, the definition of a class construct is made without presuming anything about e.g. the existence of an inheritance construct.

The semantics of a relationship is described in two, complementary ways: how the relationship modifies the set of specification objects it maps onto (its value) when the owning specification object (its origin) performs an action, and how the relationship affects the result when the owning specification object performs an action. We therefore introduce two operations for the semantics of a relationship construct: preAction and postAction. The *preAction* operation is used for specifying the effects a specific kind of Action has on the value of the relationship, while the *postAction* operation specifies what effects the Action has on the evaluation environment in which the Action is performed. Both of these operations take three arguments:

`a` the Action which is performed by the specification object owning the relationship

`e` the environment in which the Action is evaluated; basically, it is an evaluation stack where the evaluation result is pushed when a specification object performs an Action

`l` the current occurrence of the relationship; its main properties are the origin of the occurrence and its value, i.e. a collection of associated specification objects

The preAction and the postAction operations are complemented with the *consistent* operation, which specifies the invariant of the relationship, i.e. properties which must always be true for a particular relationship of that kind. The operation takes only one argument, namely an occurrence of the relationship, and to be correct from a semantical point of view, it should result in *true* for the occurrence of the construct.

Example: *Let us take a simple example to present how the preAction, postAction and consistent operations are used for specification of the semantics of a relationship construct. Pointer, which is not part of UML, is a relationship construct which can be defined between objects. A pointer has a name and a value, which is one object or a collection of objects. Basically, a Pointer relationship between two objects corresponds to an instance variable in an object-oriented programming language. The name of the relationship corresponds to the variable's name, and the target of the relationship corresponds to the value of the variable.*

The relationship construct can be specified using the technique we described above. A preAction operation states what effects an action performed by the origin of the relationship has on the relationship. A typical example of an action which may effect a Pointer is assignment, i.e. when a value is assigned to a name. When an object performs an assignment, the Pointer relationship, with a name matching that of the assignment, is modified so that from then on it maps onto the value supplied in the assignment. Since we are using an operational specification technique with an evaluation environment, the value to be assigned is the top value of the evaluation stack.

preAction (a, e, l)
```
CASE a
WHEN IN AssignValue DO
    – If the name in the action matches the name of the relationship, then set the value
    of the relationship to be the top value of the evaluation stack
    IF a name () = l name () THEN l setValue (e topValue ())
```

What effects the relationship has on the evaluation environment when an action is performed is specified with a postAction operation. Typically, this operation modifies the top value of the "stack". In our example, using the relationship's name in the owning object, i.e. the origin of the relationship, will result in the collection of objects which is the value of the relationship. Therefore, the postAction operation must state that the value of the relationship is to be added to the top value of the evaluation stack. (Note that nothing in the specification of the Pointer construct should preclude the existence of several Pointer relationships with the same name originating from the same object. Such a requirement should be expressed in the object construct instead, since some kinds of objects allow several occurrences with the same name, while other kinds of objects do not.)

postAction (a, e, l)
```
CASE a
WHEN IN LookupName DO
    – If the name is correct, then add the value to the top value of the evaluation stack.
    IF a name () = l name () THEN e appendToTop (l value ())
```

(The appendToTop *operation removes the top element from the evaluation stack, adds the argument to the removed collection, and pushes the extended collection back onto the stack.)*

Finally, we have to specify the invariant of the construct, i.e. the well-formedness requirements. In our example, we have chosen to define that each reference may appear only once in the collection.

consistent (l)

```
  – There must not be more than one occurrence of each reference in the value of a pointer.
   c := TRUE;
   FOREACH i IN l value () DO
      c := c AND (FIND j IN l value () SUCHTHAT i = j) TAIL ≠ NULL;
   c
```

The sequence of values is searched for duplicates. If the instance of the construct is valid, the operation should result in true.

This technique implies that the static semantics of a construct in UML, i.e. the well-formedness rules, is expressed with the *consistent* operation, while the dynamic semantics is expressed with the other operations: for the relationships, it is expressed with the *preAction* and the *postAction* operations. Note that the dynamic semantics of a modelling language includes the semantics used in the development of the model as well as the run-time semantics of the model. This implies that one operation, e.g. the preAction operation, will specify the semantics of both the development of the model and the execution of the model.

In the following sections we will use this approach when giving the semantics of some of the more important relationship constructs in UML.

4 Contents of Namespace

One important property of some of the constructs in UML is that they define *namespaces*, i.e. an element of such a construct can own other elements, like Classifiers and Associations. The name of an owned element must be unique within the owning namespace. Moreover, when a namespace is removed, so are its owned elements. Typically, Packages and Subsystems define namespaces, but other constructs, like Class and Node, also define namespaces. As we will see later, an element is only owned by one namespace, but it may be visible in other namespaces.

Each element which is owned by a namespace has a *visibility* property in the namespace which indicates if the element is visible outside the namespace. In other words, it indicates if the element may be referenced by elements outside the namespace. The visibility property can take one of the following three values:

public	the element is visible outside the current namespace
protected	the element is not visible outside the current namespace, except in those namespaces which have Generalization relationships to the current one
private	the element is not visible outside the current namespace

The relationship between a namespace and an element owned by the namespace is formalized as an *ElementOwnership* relationship. The relationship has a property named *visibility* which takes its value from the set of values stated above. In our specification of UML, each occurrence of a namespace has three occurrences of this kind of relationship – one for each kind of visibility.

Example: *The specification of ElementOwnership consists of the three operations* preAction, postAction *and* consistent. *The preAction operation specifies how an Action performed by the owner of the relationship affects the value of the relationship. If a new*

Classifier (or another kind of element with a name) is to be added to a namespace, each of the namespace's three ElementOwnership relationships checks if the visibility of the new Classifier matches the visibility of the set of Classifiers managed by the relationship. If it does, the new Classifier, which is the top element on the evaluation stack, is added to the set, and the extended set is the new value of the relationship.

Moreover, when an element is to be removed from the namespace, each of the three relationships iterates through its set of elements and keeps only those with a name that is distinct from the given name.

Finally, when the namespace is to be deleted, each of the three relationships iterates through their sets of elements and deletes each element.

This is expressed formally with:

```
preAction (a, e, l)
  CASE a
  WHEN IN AddElement DO
      – Add the element if the visibility matches the visibility of the relationship.
      IF a visibility () = l visibility () THEN
          l setValue (l value () ADD e topValue ())
  WHEN IN RemoveElement DO
      – Remove all elements with a matching name.
      l setValue (FIND v IN l value () SUCHTHAT v name () ≠ a name ())
  WHEN IN CeaseToExist DO
      – The namespace is about to be removed, so all its owned elements are also removed.
      FOREACH v IN l value () DO
          v ceaseToExist ();
      l setValue (NULL)
```

To specify how the relationship affects the evaluation environment when an Action is performed, we give a postAction operation. When a name is to be looked up in the namespace, each of the three relationships checks if the visibility of the wanted element should be less than or equal to the visibility of the elements managed by the relationship, i.e. it checks if the set of public elements, the set of the public and protected elements, or the set of public, protected and private elements should be searched. If the value of the relationship should be included, the relationship iterates through its set of elements and collects all elements with the desired name. Finally, the relationship modifies the existing evaluation result by adding the found set of elements to it.

If the namespace is asked for its contents, each relationship adds its set of elements to the existing evaluation result if the visibility of the contents is less than or equal to the visibility of the relationship.

```
postAction (a, e, l)
  CASE a
  WHEN IN LookupName DO
      – If the visibility required by the action is less than the visibility of the
      relationship, then add all elements with the desired name to the result.
      IF a visibility () ≤ l visibility () THEN
          e appendToTop (FIND v IN l value () SUCHTHAT v name () = a name ())
  WHEN IN Contents DO
      – If the visibility required by the action is less than the visibility of the
      relationship, then add all the elements to the result.
      IF a visibility () ≤ l visibility () THEN
          e appendToTop (l value ())
```

Finally, we state the invariant of the construct. The only requirement for an ElementOwnership relationship to be semantically well-defined is that the visibility of its managed set of elements is either public, protected or private.

consistent (l)

```
– The visibility must be one of the defined possibilities.
l visibility () = 'public' OR
l visibility () = 'protected' OR
l visibility () = 'private'
```

Together, these three operations give a formal specification of the ElementOwnership construct.

Note that the requirement that one element is owned only by one namespace is a requirement of the owned element and not of the namespace and its relationship to the element.

5 Import

In this section we will focus on the Import relationship between Packages, but it should be noted that the Import relationship can also be used between Subsystems with the same semantics. Packages are namespaces in UML, i.e. they own different kinds of Classifiers, like Classes, Interfaces and Use cases, as well as Associations, Packages and Subsystems. A Package defines a namespace for its owned elements, which means that the names of the owned elements must be unique in the Package and that each name used inside the Package must be visible in the namespace. This implies e.g. that an Association (or any other kind of relationship) may only be defined between elements that are visible in the namespace of the Association, i.e. visible in the Package which owns the Association.

The only way to define an Association between elements defined in separate Packages is by extending the namespace of one Package so that it contains the elements which should be associated to each other. This extension can be accomplished in two different ways in UML: either by a Package *importing* other Packages, or by a Package *inheriting* other Packages. In this section we will discuss the former extension and in a later section the latter.

In UML an *Import* relationship is a directed relationship from one Package to another Package, meaning that the *public* contents of the target Package is available also in the importing Package and that it may be used, e.g. in Associations, in that Package.

Note, however, that these imported elements are only referenced by the importing Package – they are not owned by it. Each element is owned by only one namespace but it may be referenced in several namespaces. Deletion of the importing Package will remove only elements owned by the importing Package. As explained in section 2, the contents of a Package includes both the elements owned by the Package and the elements referenced, i.e. those imported from other Packages.

When a Package imports another Package, all public elements contained in the imported Package extend the namespace of the importing Package. This may cause name conflicts between these elements and the elements already existing in the namespace of the importing Package (either owned by the importing Package or already imported from other Packages). To solve this problem, UML allows a referenced element to be given an alias. If this is case, the referenced element is known only under its alias in the importing Package and not under its original name.

All referenced elements are obviously public in their owning Packages. However, it is sometimes necessary to restrict the visibility of an element imported from another Package, so that it will not be exported farther. Therefore, a referenced element has a visibility property in the importing Package which is defined separately from the one of the element in the imported Package. However, this visibility cannot be greater than the visibility assigned to the element in the imported Package.

The definition of the Import construct will include the semantics of adding and removing importation of Packages. Moreover, it will also state the requirement that all visible contents of an imported Package should be available in the contents of the importing Package. The definition presumes that each Package has one instance of the construct which may connect to several Packages.

preAction (a, e, l)

```
CASE a
WHEN IN AddImport DO
    – Add the element to the relationship.
    l setValue (l value () ADD e topValue ())
WHEN IN RemoveImport DO
    – Remove the top element from the collection of imported elements.
    l setValue (l value () WITHOUT e topValue ())
```

postAction (a, e, l)

```
CASE a
WHEN IN ImportedNamespace DO
    – Add the collection of imported elements to the result.
    e appendToTop (l value ())
```

consistent (l)

```
c := TRUE;
– Each imported element must be visible in the namespace of the origin, and each public
element in the imported element must be visible in the origin.
FOREACH v IN l value () DO
    c := c AND l origin () namespace () contents ('private') INCLUDES v;
    FOREACH e IN v contents ('public') DO
        c := c AND l origin () contents ('private') INCLUDES e;
c
```

It can be noted that a Package in UML implicitly imports all its owned Packages. This implies that all elements recursively contained in a Package are visible in that Package, unless their visibility is restricted by a lower level Package.

We also need to define *ElementReference*, the relationship between a namespace and an imported element. It will be quite similar to the ElementOwnership relationship defined in section 4, but with an additional name which is used as the alias. If no alias is given then the original name is used instead.

postAction (a, e, l)

```
CASE a
WHEN IN LookupName DO
    – If the visibility required by the action is less than the visibility of the
    relationship, then add all elements with a correct alias or, if such visibiliy
    does not exist, add the correct name to the result.
    IF a visibility () ≤ l visibility () THEN
        IF (v alias () = a name ()) OR
                (v alias () = '' AND l value () name () = a name ()) THEN
            e appendToTop (l value ())
WHEN IN Contents DO
    – If the visibility required by the action is less than the visibility of the
    relationship, add all the elements to the result.
    IF a visibility () ≤ l visibility () THEN
        e appendToTop (l value ())
```

consistent (l)

```
 – The visibility must be one of the defined possibilities.
 c := l visibility () = 'public' OR
       l visibility () = 'protected' OR
       l visibility () = 'private';
 – The referenced elements must be visible in one of the imported namespaces.
 FOREACH v IN l value () DO
     c := c AND (FIND ns IN l origin () importedNamespace () SUCHTHAT
         (ns contents ('public') INCLUDES v)) ≠ NULL;
 c
```

6 Association

In UML the Association construct defines a semantic connection between the connected Classifiers, i.e. an Association is a declaration of Links. Instances of (subclasses of) the Classifiers may use the Links between them to communicate with each other and to retrieve references to the connected Instances.

An Association has a name which is composed of several parts: one is defined for the Association itself (and may be empty), and the other parts are the names of the connected Classifiers. In this paper we use the composite name, which is the one that must be unique within a namespace, as the name of an Association. Moreover, an Association consists of a collection of at least two AssociationEnds. An *AssociationEnd* connects an Association to a Classifier as well as defines the properties that end-point. It states for example:

- the *name* of the end-point, i.e. a name which is to be used for traversing the Association; the name is unique both within the Association and within each of the Classifiers of the *other* ends of the Association
- the *navigability* of the end-point, i.e. it states if a Link of the Association may be used for accessing the Instance connected to that particular end of the Link
- the *multiplicity* of the end-point, i.e. it defines to how many Instances one opposite Instance may be connected to at that end of the relationship
- the *changeability* of the end-pont, i.e. it states if Instances of the associated Classifier may be added and/or removed once an opposite Instance has been created.
- the *aggregation* of the end-point, i.e. it defines if an Instance of the associated Classifier is an aggregation of Instances of the opposite end

Furthermore, an Association may also have features of its own. In some modelling situations it is important to be able to express that part of the information is related to the relationship itself and not to the connected Instances. Therefore, UML also allows that features, like Attributes and Operations, are declared for an Association. In fact, an Association can have all the properties a Class can have; such an Association is called an Association class.

It is therefore quite natural to specify an Association in a Class-like way. The advantage of such a specification technique is that it covers the fact that an Association with a collection of Attributes is one conceptual entity and not two, as one might easily be led to believe from the notation of UML: one Association and one Class. Furthermore, we can specify all the Associations in a uniform way, regardless of whether they have any extra features.

Note that the following is not a specification of a map but of a specification object which contains a collection of maps. This collection is called *mapCollection* in the specification, and each contained map is a specification of an AssociationEnd, an Attribute or other kinds of relationship. We have only included a few of the properties of an Association, and none of the Class-like properties.

connectClassifier (c)
– Do nothing. Handled by the association-ends.

getAllNames (v)
– The association does not have any local names. Handled by the association-ends etc.
RETURN NULL

getAllEndNames ()
– The association does not have any names of the association-ends.
Handled by the association-ends.
RETURN NULL

```
getOppositeNames (c)
    – Collect the names of the association-ends which are opposite from c.
    ns := FIND n IN SELF getAllEndNames () SUCHTHAT SELF lookupType (n) = c;
    IF ns TAIL = NULL THEN
        RETURN SELF getAllEndNames () WITHOUT ns
    ELSE
        RETURN SELF getAllEndNames ()

lookupType (n)
    – Return the classifier connected to the end named n. Handled by the association-ends.
    RETURN NULL

consistent ()
    – There must be at least two association-ends.
    c := (FIND r IN mapCollection SUCHTHAT r isKindOf (AssociationEnd)) TAIL ≠ NULL;
    – Each name must be unique.
    FOREACH n IN SELF getAllNames ('private') DO
        c := c AND (FIND q IN SELF getAllNames ('private') SUCHTHAT p = q) TAIL = NULL;
    c
```

The AssociationEnd construct is specified in the same way as other kinds of relationships were specified above.

```
preAction (a, e, l)
    CASE a
    WHEN IN ConnectClassifier DO
        – If the name of the end matches the desired name, then use the value as the
        classifier of the end.
        IF a name () = l name () THEN
            l setValue (e topValue ())

postAction (a, e, l)
    CASE a
    WHEN IN GetAllNames DO
        – Add the name of the relationship to the result.
        e appendToTop (l name ())
    WHEN IN GetAllEndNames DO
        – Add the name of the relationship to the result.
        e appendToTop (l name ())
    WHEN IN LookupType DO
        – If the name of the end matches the selected name, then add the value of the
        relationship, i.e. the classifier, to the result.
        IF a name () = l name () THEN
            e appendToTop (l value ())

consistent (l)
    – The connected classifier should be connected to the association, and it should
    also be visible in the namespace of the association.
    (l value () connectedAssociations () INCLUDES l origin ())
    AND
    l origin () namespace () contents ('private') INCLUDES l value ()
```

We also need to define another relationship construct, namely the one specifying the connection from a Classifier to its Associations. A Classifier may be connected to a set of Associations, but these Associations are not necessarily owned by the same namespace as the Classifier. The specification must therefore capture that adding or removing an Association does not modify the Classifier. In our specification this is expressed with one occurrence of the relationship for each Classifier. It maps onto all the Associations connected to the Classifier.

```
preAction (a, e, l)
    CASE a
    WHEN IN AddAssociation DO
        – Add the top element to the collection of associations.
        l setValue (l value () ADD e topValue ())
    WHEN IN RemoveAssociation DO
        – Remove the top element from the collection of associations.
        l setValue (l value () WITHOUT e topValue ())

postAction (a, e, l)
    CASE a
    WHEN IN GetAllNames DO
        – Collect all opposite names from the associations.
        FOREACH v IN l value () DO
            n ADD v getOppositeNames (l origin ());
        e appendToTop (n)
    WHEN IN ConnectedAssociations DO
        e appendToTop (l value ())
    WHEN IN LookupType DO
        FOREACH v IN l value () DO
            t ADD v lookupType (a name ());
        e appendToTop (t)

consistent (l)
   – Each connected association must have at least one end connected to the classifier
   and each association must appear only once in the collection.
    c := TRUE;
    FOREACH v IN l value () DO
        c := c AND
            (FIND n IN v getAllEndNames () SUCHTHAT v lookupType (n) = l origin ()) ≠ NULL;
    FOREACH v IN l value () DO
        c := c AND (FIND o IN l value () SUCHTHAT v = o) TAIL ≠ NULL;
    c
```

Due to the scope of this paper we have not included the descriptions of the Link and the Link-End constructs. These specifications corresponds to the specifications of the Association and the AssociationEnd constructs.

7 Different Kinds of Generalization

One of the more fundamental object-oriented constructs is that a class can inherit features from other classes, and different semantics have been proposed in different languages. In UML the *Generalization* relationship is used for expressing inheritance of features. Generalization is a taxonomy relationship which is defined between a more general element and a more specific one. The more specific element is compatible with the more general one in the sense that all features defined in the general element appear in the specialized element. Since UML contains several different kinds of constructs which are generalizable it is not possible to define the Generalization relationship in only one way, however; instead its semantics will vary from case to case depending on what kind of associated elements are involved. For example, the semantics of a Generalization relationship between Classes differs slightly compared to a Generalization relationship between Packages.

Still, some of the semantics is common to all the different kinds of Generalization constructs in UML:

- a Generalization relationship may only be defined between elements of the same kind
- a Generalization hierarchy must not be cyclic
- an element which appears more than once in a Generalization hierarchy (which is possible since multiple Generalizations from one element is allowed) is still inherited only once

- all features of the more general element must also appear in the specialized element, although possibly in a more specialized form
- the target of the Generalization must not have been defined as a leaf in the hierarchy, and the source of the Generalization must not have been defined as a root

Most of the different kinds of semantics of Generalization defined in UML will be described in the following subsections.

7.1 Generalization between Classes

A Generalization between Classes implies that an Object of the subclass can always be used whenever an Object of the superclass is expected. This means that an Object, which is an instance of a Class, contains Attribute links corresponding to all the Attributes defined in the Class as well as in all its superclasses. In the same way, the Object participates in Links corresponding to the Associations connected to the Class and all its superclasses. Moreover, all the Operations defined in the Class and its superclasses are applicable to the Object. This substitutability principle leads to the following requirements:

- the subclass must provide at least the same set of Operations as the superclass, although possibly in a more restricted form, i.e. the parameter types may be generalized and the return types may be specialized
- the subclass must provide at least the same set of opposite AssociationEnds and Attributes as the superclass; note that they must not be restricted in the subclass (see e.g. [Bru96] for a thorough discussion of this subject)
- the elements in the namespace of the superclass must also be visible in the namespace of the subclass

An Object contains sets of features corresponding to the features declared in its Class and in its superclasses. A straight-forward way of specifying an Object is therefore to introduce segments, one for each of these sets. A *segment* contains all the features of an Object which are declared in one Class. One Object will therefore consist of one or more segments – one for each Class in the Generalization hierarchy of the originating Class.

The definition of the Generalization relationship between Classes together with the notion of segments results in the following specification of the semantics:

```
preAction (a, e, l)
  CASE a
  WHEN IN AddGeneralization DO
      – Add the element to the relationship.
      l setValue (l value () ADD e topValue ())
  WHEN IN RemoveGeneralization DO
      – Remove the element from the relationship.
      l setValue (l value () WITHOUT e topValue ())

postAction (a, e, l)
  CASE a
  WHEN IN CreateSegment DO
      – Create a segment of each superclass and add the collection to the segments
      already created by the subclass unless such a segment is already included.
      r := e popResult ();
      FOREACH v IN l value () DO
         IF (FIND s IN r SUCHTHAT s class () = v) = NULL THEN
            r ADD v createSegment ();
      e pushResult (r)
```

```
WHEN IN GetNamedOperation DO
    – Collect all the operations from the superclasses with the correct name and add
    the collection to the subclass' operation.
    IF a visibility () ≠ 'private' THEN
        r := e popResult ();
        FOREACH v IN l value () DO
            s := v getNamedOperation (a name (), a visibility ());
            IF NOT r INCLUDES s THEN
                r ADD s;
        e pushResult (r)
WHEN IN GetNamedStructuralFeature DO
    – Collect all the attributes, opposite association-ends and the like with the correct
    name from the superclasses and add them to those retrieved from the subclass.
    IF a visibility () ≠ 'private' THEN
        r := e popResult ();
        FOREACH v IN l value () DO
            s := v getNamedStructuralFeature (a name (), a visibility ());
            IF NOT r INCLUDES s THEN
                r ADD s;
        e pushResult (r)
WHEN IN GetAllNames DO
    – Add all the names of the structural features of the superclass to the result.
    IF a visibility () ≠ 'private' THEN
        r := e popResult ();
        FOREACH v IN l value () DO
            n := v getAllNames (a visibility ());
            IF NOT r INCLUDES n THEN
                r ADD n;
        e pushResult (r)
WHEN IN LookupName DO
    – Collect all the non-private elements with the desired name and add them to the result.
    IF a visibility () ≠ 'private' THEN
        r := e popResult ();
        FOREACH v IN l value () DO
            i := v lookupName (a name (), a visibility ());
            IF NOT r INCLUDES i THEN
                r ADD i;
        e pushResult (r)
WHEN IN Contents DO
    – Collect all the non-private elements and add them to the result.
    IF a visibility () ≠ 'private' THEN
        r := e popResult ();
        FOREACH v IN l value () DO
            i := v contents (a visibility ());
            IF NOT r INCLUDES i THEN
                r ADD i;
        e pushResult (r)
WHEN IN IsInGeneralizationHierarchy DO
    – Check if the top value is in (the transitive closure of) the relationship.
    i := FALSE;
    FOREACH v IN l value () DO
        i := IF v = e topValue () THEN TRUE
            ELSE i OR v isInGeneralizationHierarchy (e topValue ());
    e pushResult (i)
```

consistent (l)

```
  – The subclass and the superclasses must be some kind of classes.
  c := l origin () isKindOf (Class);
  FOREACH v IN l value () DO
      c := c AND v isKindOf (Class);
  – The superclasses must be visible in the namespace of the subclass.
  FOREACH v IN l value () DO
      c := c AND l origin () contents ('private') INCLUDES v;
  – The subclass must not appear in the generalization hierarchy.
  FOREACH v IN l value () DO
      c := c AND NOT v isInGeneralizationHierarchy (l origin ());
```

```
– The subclass must not be defined to be a root, and the superclass must not be
defined to be a leaf.
FOREACH v IN l value () DO
    c := c AND NOT v isLeaf ();
c := c AND NOT l origin () isRoot ();
– Each operation in the subclass with the same name and number of parameters as
an operation in the superclass the subclass' operation must be a specialization of
the superclass' operation.
FOREACH o IN l origin () getOperations () DO
    FOREACH v IN l value () DO
        s ADD v getNamedOperation (o name (), 'protected');
    FOREACH i IN s DO
        IF i noOfParameters () = o noOfParameters () THEN
            c := c AND o specializationOf (i);
– No structural feature can have the same name as a structural feature in
the superclass.
FOREACH p IN l origin () getStructuralFeatures () DO
    FOREACH v IN l value () DO
        c := c AND v getNamedStructuralFeature (p name (), 'protected') = NULL;
c
```

It is to be noted that the semantics of the Generalization relationships between Interfaces and between Data types are subsets of the semantics of the Generalization relationship between Classes. Interfaces and Data types can define only a set of Operations but no other kinds of features. Moreover, they do not act as namespaces for other elements. The specification of these two Generalization relationships will be the same as the one above, but with obvious modifications.

It should also be pointed out that Generalization between Associations, i.e. between Association classes, is also covered by this specification, since the AssociationEnds are treated as structural features of an Association.

7.2 Generalization between Packages

A Package has no Associations, Operations or Attributes of its own. However, the contents of a Package is inherited by specializing Packages. This means that all the elements contained in a Package, i.e. both the owned and the referenced elements, which are either public or protected, are visible inside the namespace of the specializing Package. However, a specializing Package may not define new aliases or visibilities for the inherited elements. Therefore, when a name is used inside a Package the, public and protected contents of the namespaces of the inherited Packages are also searched.

```
postAction (a, e, l)
    CASE a
    WHEN IN AddGeneralization DO
        – Add the element to the relationship.
        l setValue (l value () ADD e topValue ())
    WHEN IN RemoveGeneralization DO
        – Remove the element from the relationship.
        l setValue (l value () WITHOUT e topValue ())
```

```
postAction (a, e, l)
    CASE a
    WHEN IN LookupName DO
        – Collect all the non-private elements with the desired name and add them to the result.
        IF a visibility () ≠ 'private' THEN
            r := e popResult ();
            FOREACH v IN l value () DO
                i := v lookupName (a name (), a visibility ());
                IF NOT r INCLUDES i THEN
                    r ADD i;
            e pushResult (r)
```

```
WHEN IN Contents DO
    – Collect all the non-private elements and add them to the result.
    IF a visibility () ≠ 'private' THEN
        r := e popResult ();
        FOREACH v IN l value () DO
            i := v contents (a visibility ());
            IF NOT r INCLUDES i THEN
                r ADD i;
        e pushResult (r)
WHEN IN IsInGeneralizationHierarchy DO
    – Check if the top value is in (the transitive closure of) the relationship.
    i := FALSE;
    FOREACH v IN l value () DO
        i := IF v = e topValue () THEN TRUE
            ELSE i OR v isInGeneralizationHierarchy (e topValue ());
    e pushResult (i)
```

consistent (l)

```
    – The subclass and the superclasses must be some kind of Packages.
    c := l origin () isKindOf (Package);
    FOREACH v IN l value () DO
        c := c AND v isKindOf (Package);
    – The superclasses must be visible in the namespace of the subclass.
    FOREACH v IN l value () DO
        c := c AND l origin () contents ('private') INCLUDES v;
    – The subclass must not be defined to be a root, and the superclass must not
    be defined to be a leaf.
    FOREACH v IN l value () DO
        c := c AND NOT v isLeaf ();
    c := c AND NOT l origin () isRoot ()
    – The subclass must not appear in the generalization hierarchy.
    FOREACH v IN l value () DO
        c := c AND v isInGeneralizationHierarchy (l origin ());
    c
```

7.3 Generalization between Subsystems

A Subsystem is a kind of Package with an additional specification part of the behaviour performed by Instances of the Classifiers contained in the Subsystem. Typically, the specification part contains a set of Use cases, but the Subsystem construct also makes it possible for us to use Operations for the specification of its contained behaviour. Moreover, a Subsystem has no Attributes of its own, and an Association connected to a Subsystem is actually connected to an element contained in the Subsystem.

This means that the specification of a Generalization relationship between Subsystems is the same as the corresponding specification between Packages (with the obvious modifications). However, we need also to include the specification of inheriting Operations as is defined for Generalizations between Classes.

7.4 Generalization between Use cases

Use cases are used for defining the behaviour of an element, e.g. a system, without revealing anything about the element's internal structure. A Use case specifies a set of sequences of Actions which the element specified by the Use case can perform, interacting with the users of that element. A Use case is a kind of Classifier, so it can have Attributes as well as Associations to other Classifiers, but these should exist outside the element which is specified by the Use case. However, a Use case must not contain other elements.

Two different kinds of Generalization relationships exist between Use cases: Uses and Extends. In this paper we will discuss only Uses since its semantics corresponds to the Generalization relationships discussed above. A *Uses* relationship between two Use cases implies that the sequence of Actions in the specialized Use case includes the sequence of Actions of the generalized Use case. The specialized Use case may have additional Actions added to the sequence – not only at the beginning or at the end, but anywhere in the sequence. However, it must not re-order the inherited sequence in any way. Moreover, the parameter types and the return types of the Operations of a Use case must not be modified (cf. Generalization between Classes).

The specification of the Uses construct is quite similar to that of Generalization between Classes, with the exceptions of contained elements and specialization of Operations, and with the additional requirement of keeping the order of the inherited sequences.

8 Concluding Remarks

In this paper we have presented some of the results from our work developing a formal specification of UML. The focus has been on the major relationship constructs of UML, which we have given formal and also more detailed specifications compared to those of the language definition [USE97].

There are other kinds of relationships in UML which have not been covered in this paper, e.g. *Dependency* and a set of special cases of the Dependency relationship, like *Refinement*. All these kinds of relationships express that a client is somehow dependent on a supplier. The semantics of these constructs has (so far and intentionally) been left rather undefined in UML, and we have therefore not given them a formal specification.

Constraint is a construct which is used for expressing a condition or restriction between different elements. This construct is easily expressed with our technique, since the description of the Constraint is a kind of invariant. It must be fulfilled for a system to be well-formed. The body of the Constraint is simply expressed in the consistent operation.

Some of the relationship constructs covered in this paper are in fact Stereotypes on other constructs: the Import relationship is a Stereotype on the Dependency relationship, and the Uses relationship a Stereotype on the Generalization relationship.

The fact that certain constructs in UML lack a complete and detailed semantic definition has made it impossible to provide formal specifications to all UML constructs without modifying the definition of the language. In some cases, the language definition may be modified, but in other cases it must not be changed. Some parts of the semantics of a modelling language can and should rely on the intuition of the modeller. In this way, the modeller will be able to express his thoughts, without being hampered by details which can be dealt with later.

We hope that the deficiencies in the definition of the language will be removed in future versions of the language, and we believe that a good way of finding these deficiencies is by providing a formal specification of the language. The usage of a language like ODAL, with its formal semantics, is suitable for the specification of both the well-formedness rules and the dynamic semantics of UML. The current specification of UML is done partly in OCL and partly in natural language. OCL is well-suited for expressing well-formedness rules, but a specification of the dynamic semantics expressed with constraints would not be easily understood. However, since OCL lacks a formal specification of its semantics, it is unsuitable for formalizing UML.

The specification technique used in this project is a general technique for specification of object-oriented modelling languages. We have previously used it for specification of a subset of OOSE [JCJÖ93] where the technique proved to be suitable, not only for the development of the specifications but also for later modifications.

References

[BHK+97] R. Breu, U. Hinkal C. Hofmann, C. Klein, B. Paech, B. Rumpe, and V. Thurner. Towards a Formalization of the Unified Modeling Language. In M. Aksit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming, ECOOP'97, Lecture Notes in Computer Science 1241*, pages 344–366. Springer-Verlag, June 1997.

[Boo91] G. Booch. *Object-Oriented Design with Applications.* Redwood City, 1991.

[Bru96] K. B. Bruce. Typing in object-oriented languages: Achieving expressiveness and safety. Technical report, Department of Computer Science, Williams College, MA, USA, 1996.

[HG97] D. Harel and E. Gery. Executable Object Modeling with Statecharts. *IEEE Computer*, 30(7):31–42, July 1997.

[HN96] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.

[JCJÖ93] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, 1993.

[Jon93] C. B. Jones. A pi-calculus Semantics for an Object-Based Design Notation. In E. Best, editor, *CONCUR'93: 4th International Conference on Concurrency Theory Lecture Notes in Computer Science 715.* Springer-Verlag, 1993.

[KR97] H. Kilov and B. Rumpe. ECOOP'97 Workshop on Precise Semantics for Object-Oriented Modeling Techniques. Technical Report TUM-I9725, Institut für Informatik, Technische Universität, München, Germany, 1997.

[LO96] M. Loomis and J. Odell. *Object Analysis & Design RFP-1.* Object Management Group, Framingham Corporate Center, 492 Old Connecticut Path, Framingham MA 01701-4568, 1996. OMG TC Document ad/96-05-01 – rev.1, http://www.omg.org/docs/ad/96-05-01.ps.

[MPW92] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, I. *Information and Computation*, 100:1–40, 1992.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, Englewood Cliffs, 1991.

[RWL96] T. Reenskaug, P. Wold, and O. A. Lehne. *Working with Objects: The OOram Software Engineering Methodology.* Manning Publications, 1996.

[SF97] M. Shroff and R. B. France. Towards a Formalization of UML Class Structures in Z. In *Proceedings of Twenty-First Annual International Computer Software and Applications Conference (COMPSAC'97)*, pages 646–651. IEEE COmputer Society, 1997.

[SGW94] B. Selic, G. Gullekson, and P. Ward. *Real-Time Object-Oriented Modeling.* John Wiley and Sons, 1994.

[UML97] *UML Proposal to the Object Management Group, version 1.1*, September 1997. http://www.rational.com/uml/1.1/.

[USE97] *UML Semantics, version 1.1*, September 1997. Part of [UML97].

[Wal95] D. Walker. Objects in the π-Calculus. *Information and Computation*, 116:253–271, 1995.

Appendix

ODAL is a simple, strongly and statically typed object-oriented language. It uses the key ideas of object-orientation and combines the specification of the structure and the dynamics of a specification entity into a single unit. To create, understand or adjust the definition of an entity, one unit in the entire specification is developed, read or modified. The object-oriented technique also makes it possible to express features such as commonalities between different units (inheritance), concurrency between units (objects), and clean interfaces between units (encapsulation and type).

ODAL has an object construct and a class construct. The objects execute concurrently, and both send-and-wait and send-no-wait semantics are defined for communication. Each object has a set of instance variables holding either zero or one, or a (possibly empty) sequence of references to objects. These variables are used for expressing the relationships between the objects. Each object originates from a class which also defines the type of the object.

ClassDef	::	CLASS ClassName [SUPERCLASS ClassName] [VARIABLES VarDef*] [METHODS MethDef*]
VarDef	::	VarName : Type
Type	=	ClassName | BOOLEAN | Type*
MethDef	::	MethodName ([VarDef*]) : Type [| VarDef* |] Expr
Expr	=	ExprSeq | AssignExpr | CondExpr | MsgExpr | IterExpr | FindExpr | ...
ExprSeq	::	Expr ; Expr
AssignExpr	::	VarName := Expr
CondExpr	::	IF Expr_b THEN Expr_t [ELSE Expr_f]
MsgExpr	::	Expr MethodName ([Expr_i*])
IterExpr	::	FOREACH VarName IN Expr_s DO Expr
FindExpr	::	FIND VarName IN Expr_s SUCHTHAT Expr_b

Figure 1. The syntax of some of the ODAL *constructs.*

The meaning of most of these expressions is quite obvious and we will therefore only explain the last two of them. An *iteration* expression involves two steps. First the $Expr_s$ expression is evaluated. Then, the *Expr* expression is evaluated for each of the resulting values. The current value is available during the evaluation of the *Expr* expression under the name of *VarName*. The value of the complete expression is the same as the value of the last evaluation of the sub-expression. The meaning of a *find* expression is similar, but its result is the subset of the values for which the $Expr_b$ evaluates to true.

In ODAL the classes act as object templates which are used for creating objects. Each class declares what instance variables and methods should be contained in the object originating from that class. Single inheritance is allowed between classes.

The body of a method consists of a sequence of expressions, such as condition, iteration and assignment, as well as expressions for communication and object creation. Each expression evaluates to an object.

The exact interpretation of ODAL is specified in the π-calculus [MPW92], which has proved to be useful for specification of object-oriented languages [Jon93, Wal95]. The result is a new, foreceful specification language characterized by its high level of abstraction as well as its access to the theory of the π-calculus via the mapping from the ODAL language constructs to the π-calculus expressions.

$$
\begin{aligned}
&[[expr\ \mathrm{NAME}\ (expr_1 \ldots expr_n)]](e,r,\ldots) \stackrel{\mathrm{def}}{=} \\
&\quad (\nu c, c_1 \ldots c_n, rc, mc, vc, ac) \\
&\quad [[expr]](e,c,\ldots) \mid [[expr_1]](e,c_1,\ldots) \mid \ldots [[expr_n]](e,c_n,\ldots) \mid \\
&\quad c(rec)\ .\ c_1(a_1)\ldots c_n(a_n)\ .\ \overline{rec}(rc)\ .\ \overline{rc}(mc,vc)\ .\ \overline{mc}(name,ac)\ .\ ac(mi)\ .\ \overline{mi}(a_1\ldots a_n,r)
\end{aligned}
$$

Figure 2. In the mapping onto π-calculus expressions, each ODAL *construct is evaluated in an environment, e, and results in an object which is passed on to the next expression, r. The meaning of a send expression is: the receiver sub-expression and the argument sub-expressions are evaluated in parallel and, after the results of the evaluations have been collected, the encoding of the matching method is retrieved from the receiver. Finally, the arguments and the information about where the evalution is to continue after the evaluation are passed on to the method which starts its execution.*