Formal Specification of Object-Oriented Meta-modelling

Gunnar Övergaard

Royal Institute of Technology, Stockholm, Sweden

Abstract. Modelling languages such as the Unified Modeling Language are used during the early phases of system development to capture requirements and to express high-level designs. Many such languages have no universally fixed interpretations since different development projects often use key concepts, like Class, Generalization and Association, in slightly different ways. Therefore *meta-modelling*, i.e. the precise specification of the concepts used in a model, is of importance in order to avoid misunderstandings.
The BOOM framework, presented in this paper, is intended for this kind of meta-modelling. The framework consists of a collection of modelling constructs specified with a small object-oriented language. The framework is simple enough for an engineer to adjust the modelling concepts to project specific needs. It includes all necessary aspects of language specification, among them definition of abstract syntax, well-formedness rules, and dynamic semantics. To demonstrate its use, this paper includes a specification of some of the constructs defined in the Unified Modeling Language.

1 Introduction

There are many object-oriented system development methods used by industry today [5, 15, 17, 6, 1, 16]. Since these languages are used in the early phases of system development, expressiveness is more important than precision. Therefore, the languages typically have a rich and well-specified graphical syntax but no rigorous semantics. Some of these languages even include constructs for modifying the semantics of the language: the Unified Modeling Language [10] includes the Stereotype construct which enables the developer to modify the semantics of a construct as well as its notation. Only very few of the development methods have a formal specification, and these methods are usually developed for a specific organisation or a specific type of application, like SDL [4]. Methods that have received wider acceptance are usually informally specified.

However, informally and incompletely defined languages have several drawbacks. The system specifications in such languages will not have unique interpretations. A model can unintentionally be interpreted differently by different people, both within the project and outside the project. It is also hard to envisage effective computer tools that support the system development process, if the tool cannot access the intended meaning of the models.

T. Maibaum (Ed.): FASE 2000, LNCS 1783, pp. 193–207, 2000.
© Springer-Verlag Berlin Heidelberg 2000

To avoid these problems, a modelling language needs a kind of formal semantics which is easy to understand and which can be easily modified. The definition must admit changes in the structure of the language constructs, in the meaning of these constructs, and in the combination of the constructs. The definition must be presented in an accessible way, so that system developers can understand and adjust it at need – it has to be a flexible language. At the same time, the specification must be rigorous enough to avoid unnecessary ambiguities.

Several different formal specification methods exist today (see e.g. the *Formal Methods* home page at *http://www.comlab.ox.ac.uk/archive/formal-methods.html*). These methods are based upon different kinds of formalisms. During the last decade, object-orientation has influenced the formal community resulting in formal object-oriented methods [2, 18]. In these methods, precision is combined with the advantages of object-orientation, such as encapsulation and inheritance.

A major reason for choosing an object-oriented formal specification language instead of a traditional one is that the intended user is acquainted with the object-oriented paradigm. In this way, there is no conceptual shift between the object-oriented *modelling language* and the object-oriented *specification method*. Moreover, just as an object-oriented technique, including factoring out commonalities, encapsulation and localization of information, is preferable for system development, it is also appropriate for language specification. Traditionally, the abstract syntax of a construct is separated from the definition of the rule that state when the construct is well-formed. Moreover, the specification of the dynamic semantics of the language is often separated from the rest of the specification. Understanding or adjusting the definition of a construct given in a traditional specification will therefore be much more difficult than in a specification using an object-oriented technique, since in the traditional specification the complete definition of the construct is spread out over several places.

Another advantage with an object-oriented specification technique is that it may include a library of predefined components in a language specification. Since different object-oriented modelling languages use similar constructs, each new specification does not have to start from scratch. Deviations from the predefined definition can be expressed in subclasses of the components.

Together the components form a so-called meta-model, i.e. they constitute a model of an object-oriented modelling language. The different kinds of constructs in the language are expressed with classes in the meta-model, and the associations between these classes state the relationships between the constructs. The methods that are defined in the classes in the meta-model specify both the well-formedness rules and the dynamic semantics of the language.

In this paper we present a meta-model for specification of modelling languages. The purpose of the meta-model is to provide a framework for defining the different constructs in such a language. The idea of this kind of framework arose from our participation in the team developing and enhancing the Unified Modeling Language. This paper is organized as follows: The next section presents a framework for specification of modelling languages, while Section 3 demonstrates its usage by expressing some of the constructs defined in the Uni-

fied Modeling Language in the framework. This paper ends with some concluding remarks.

2 Boom – A Framework for Formal Specification

In this section we present Boom, a framework for specification of object-oriented modelling languages. It includes all necessary aspects of language specification, among them definition of abstract syntax, well-formedness rules, and dynamic semantics. The framework consists of a meta-model of such languages and a formal specification language called Odal which is used for defining the classes in the meta-model. Odal is a simple, strongly typed object-oriented language with a familiar syntax. Its semantics is specified using the π-calculus [8]. In this it follows the principles from e.g. Walker [19] and Jones [7] where it is shown how the π-calculus is used to define object-oriented programming languages. The complete semantics of Odal is approximately 50 pages and is out of the scope of this paper.

In Boom we adopt a *localization principle* that implies a specification technique 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, among other things, an object has a set of relationships, but the specification of the construct does not include any of the semantics of these relationships. 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 any assumptions about e.g. the existence of a generalization construct.

Boom includes all meta levels in one model, i.e. different meta levels are not separated into different models. This enables Boom to express languages that support meta-modelling, like languages that include a meta-class construct or that allow explicit relationships to cross meta levels.

It should be noted that Boom is not a model of a CASE-tool. The purpose of Boom is to specify the semantics of modelling languages, i.e. its purpose is not to provide a model of how such a language should be implemented. There are several aspects which are not considered during the development of Boom but which must be included in the design of a tool, like efficiency, storage etc.

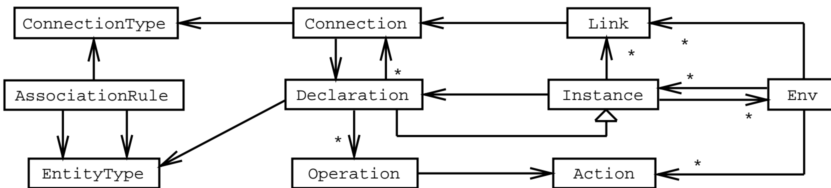

Fig. 1. A subset of Boom in UML notation.

The core of BOOM contains approximately 30 classes for defining class-like constructs, binary relationships, instances and links, i.e. each class defines the abstract syntax as well as the static and the dynamic semantics of a construct. These classes are accompanied by a set of classes for specification of operations and actions. Moreover, BOOM contains classes for stating rules how different constructs may be connected to each other. Together, these classes form a framework for specification of modelling languages.

If BOOM does not contain a class which specifies the desired semantics exactly, a new class can be added. This new class usually becomes a subclass of an already existing class in BOOM, modifying selected parts of the existing class by overriding or extending some of its definitions.

Below we describe parts of BOOM. In Section 3, we use BOOM for specification of some of the constructs in the Unified Modeling Language [10].

2.1 Class-like Constructs

In BOOM class-like constructs in a modelling language, such as a Class construct or a Data Type construct, are expressed with the class Declaration. It specifies a construct which declares a set of features, and may create instances that offer these features. There may both be structural features, like attributes and associations to other class-like constructs, as well as be behavioural features, like operations and methods. In BOOM structural features are basically expressed with the Connection class, which specifies binary, directed relationships between Declarations, while the Operation class in BOOM is used for specifications of behavioural features. Hence, a Declaration includes a set of Connections and a set of Operations. In accordance with the localization principle a Declaration depends only on the existence of its Connections and Operation, not on their specific kinds. The semantics of the different kinds of features are specified within (subclasses of) the Connection class and the Operation class.

The different kinds of instance constructs in a model, like Object and Data Value, are expressed as instances of the BOOM class Instance. In an object-oriented model, an instance is created by, and therefore follows the declaration in, a class-like construct. This implies that an Instance is associated with a Declaration (its creator, or origin), and that the Instance has a set of Links corresponding to the set of Connections declared in its Declaration, as well as that it can perform the behaviour declared in the Operations of its Declaration.

Since BOOM defines all meta levels within one model, a Declaration must be an instance of another Declaration. The class Declaration is therefore a subclass of Instance. This implies that relationships between class-like constructs, like generalization between classes, are expressed with Links between Declarations.

The detailed specifications of Declaration, Instance and their subclasses are made in ODAL, a formal object-oriented language. Due to the scope of this paper we have excluded most of these details. However, some examples are to be found in the following. Although ODAL has not been presented in this paper, we believe that a reader who is familiar with object-oriented languages will have no problem of understanding these examples.

```
CLASS Instance
VARIABLES
  origin : Declaration,
  links : Link*
INVARIANT
  origin ≠ NULL
METHODS
initialize (o : Declaration,
            c : Connections) : Instance
  origin := o;
  FOREACH con IN c DO
    links ADD con createLink ()
entityType () : EntityType
  origin instanceKind ()
evaluateAction (a : Action) : Boolean
  ...
      ⋮
```

```
CLASS Declaration SUPERCLASS Instance
VARIABLES
  operation : Operation*,
  connection : Connection*,
  instanceType : EntityType,
  name : Name
INVARIANT
  name ≠ NULL AND instanceType ≠ NULL
METHODS
addConnection (c : Connection) : Boolean
  IF SELF lookupConnection (c name ()) = NULL AND
    ruleset allowedConnection (
        SELF instanceKind (),
        c target () instanceKind (),
        c connectionType ()) THEN
      connection ADD c; TRUE
  ELSE FALSE
lookupConnection (n : Name) : Connection
  COLLECT c IN connection SUCHTHAT c name () = n
instanceKind () : EntityType
  instanceType
createInstance () : Instance
  instanceType source () NEW (SELF, connection)
      ⋮
```

For example, the meaning of adding a new Connection to a Declaration is defined in the *addConnection* operation in the Declaration class. It states that two conditions must be fulfilled: there must not be another Connection within the Declaration with the same name as the new one, and this kind of Connection must be allowed between the kinds of instances represented by the Declaration and by the target of the Connection. (The *ruleset* referenced in the operation specifies the allowed combinations. See Section 2.3.) If these conditions are fulfilled the Connection is added and the operation results in *true*.

2.2 Relationships

Boom defines binary directed relationships, called Connections, between Declarations, and corresponding binary directed relationships, called Links, between Instances. A Link can only exist between two Instances if a corresponding Connection is declared between their Declarations (cf. the declaration of a variable and the variable slot). Hence, such relationship constructs, like Pointers, are easily specified using (a subclass of) the Connection class.

```
CLASS Connection
VARIABLES
  name : Name,
  target : Declaration,
  instanceType : ConnectionType
INVARIANT
  target ≠ NULL AND name ≠ NULL AND
  instanceType ≠ NULL
METHODS
target () : Declaration
  target
connectionType () : ConnectionType
  instanceType
createLink () : Link
  Link NEW (SELF)
      ⋮
```

```
CLASS Link
VARIABLES
  origin : Connection
  value : Instance*
INVARIANT
  origin ≠ NULL
METHODS
connectionType () : ConnectionType
  origin connectionType ()
      ⋮
```

If the modelling language contains more complex relationship constructs, like N-ary relationships or bi-directional relationships, a more complex mapping onto Boom is required. Since the semantics of an N-ary relationship is in many ways similar to a class (each of its links is connected to a collection of instances), and a bi-directional relationship can be seen as a special case of an N-ary relationship, it is therefore not surprising that they are expressed in the same way as a class. For example, the different end-points of the relationship must have unique names which correspond to the requirement that the attributes of a class must have unique names. Hence, such a relationship is specified by mapping the relationship itself onto a Declaration, and the relationship's end-points onto Connections included in the Declaration (see the example in Section 3.2.)

2.3 Mapping Language Entities onto Boom Classes

Boom uses an explicit mapping between the names of the class-like and object-like constructs in a modelling language and the Boom classes that specify the constructs. There are a few reasons for doing this. First, Boom contains a set of predefined classes that have already been given names. When there is a mismatch between the name used in the modelling language and the name used in Boom, a mapping resolves the conflict. Second, several constructs may be specified with the same Boom class, e.g. a N-ary relationship may have the same semantics as an ordinary class. Instead of duplicating the Boom class, both Class and N-ary Relationship may be mapped onto the same Boom class. Third, to be able to compare constructs in different languages the names of the Boom classes should not interfere. Finally, once the mapping of the language constructs onto Boom classes has been established, the names used in the modelling language can be used. If, for example, the semantics of a construct is later modified, only the mapping has to be changed (possibly after adding a new class to Boom which includes the changes.)

The mapping is done using three classes called EntityType, ConnectionType and LinkType. Each of them pairs a modelling construct name and a Boom class; EntityType pairs the name with (a subclass of) the class Instance, while ConnectionType pairs the name and (a subclass of) the class Connection, and LinkType pairs the name with (a subclass of) the class Link. To express that a new entity is being defined in the modelling language, the class that the modelling constructs maps onto, is instantiated.

For example, assume that the Class construct has the semantics as defined by the Declaration class in Boom, and that the semantics of the Object construct is specified by the Instance class. Moreover, assume that each object can have a set of pointers to other objects, i.e. the language includes a Pointer construct and therefore also a PointerDeclaration construct; the semantics of the Pointer construct is assumed to be defined by the Link class while the PointerDeclaration construct is specified with the Connection class. The mapping of these modelling constructs onto Boom is defined by the following statements:

```
mapping ADD (EntityType NEW (CLASS, Declaration));
mapping ADD (EntityType NEW (OBJECT, Instance));
mapping ADD (ConnectionType NEW (POINTERDECLARATION, Connection));
mapping ADD (LinkType NEW (POINTER, Link));
```

When a new object is to be created, the Boom class which the name OBJECT maps onto, i.e. Instance, is instantiated.

Boom also requires that all allowed combinations of kinds of connections and kinds of declarations are explicitly enumerated. In this way it is stated what kinds of connections are meaningful between different kinds of constructs; no other combinations are allowed. For this purpose Boom uses an explicit set of association rules. Each rule is expressed with a tuple of entity and connection names: ⟨*kind of source, kind of target, kind of connection*⟩.

In our small example, classes can declare pointers to classes. We therefore state that a class can be connected to a class with a pointer declaration. This is done with the following expression:

```
ruleset ADD (AssociationRule NEW (CLASS, CLASS, POINTERDECLARATION));
```

If a declaration of a connection is added to the model which does not conform to any of the association rules, the model is not well-formed, i.e. its semantics is not defined.

After defining how the different language constructs are mapped onto the Boom classes that specify their semantics, and after stating what connections may be used between the different constructs, we can now define how the constructs in the language are connected to each other. We are using a graphical notation to do this. Figure 2 shows the different language constructs we have defined in our small example.

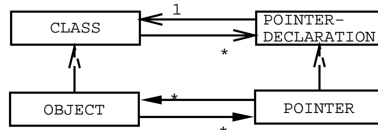

Fig. 2. A fragment of a meta-model of our example language.

A box denotes the language construct with the name equal to the string inside the box. An arrow from one box to another implies that an occurrence of the construct denoted by the source box contains a set of references to occurrences of the construct denoted by the target box. A declaration of a relationship is denoted by an open arrow head, while a closed arrow head denotes the connection itself. The multiplicity of the contained set is defined by the number or the interval at the arrow head ('*' denotes unlimited, i.e. any number of instances is allowed). A dashed arrow denotes an *instance-of* relationship.

The multiplicities stated on in these diagrams are included in the definition of the corresponding Boom classes. The multiplicity on an arrow:

- from a Declaration to a Connection is included in the invariant of the Declaration
- from a Connection to a Declaration is always '1' (cf. there is only one type in a variable declaration)
- from an Instance to a Link is always '*', since the actual number of Links is always determined by the multiplicity on the arrow between the corresponding Declaration and Connection (see also the specification of the Instance class in Section 2.1).
- from a Link to an Instance is included in the invariant of the Link

Hence, in our small meta-model we have stated that an object may have a set of pointers, and that each pointer may reference a set of objects. Furthermore, an object is an instance of, i.e. it originates from, a class. A class may declare a set of pointers, and each of these references a class.

3 Formal Specification of UML Using Boom

In this section we exemplify how Boom is used for specification of modelling languages by presenting how some of the constructs in the Unified Modeling Language (UML) [10] can be expressed in Boom. We will not present all the details regarding the Boom classes but concentrate on the usage of Boom. More detailed descriptions of the semantics can be found in e.g. [12, 11, 13]. In this paper we will focus our presentation on a few constructs in UML: Class, Association and Package.

3.1 Class Construct

In UML a class is a description of a set of objects that share the same structure and behaviour. More specifically, a class declares a set of attributes, and a set of operations, and may be attached to a collection of associations. (Due to the scope of this paper we ignore the other constructs for specification of behaviour, such as Methods and State Machines.) In Boom the Declaration class specifies precisely this: it has a name, contains a set of Operations and a set of Connections. Hence, we use Declaration for specification of the Class construct.

Each attribute of the class declares a name and a reference to a data type. In Boom the Connection class specifies a named connection to a (subclass of) Declaration. However, an attribute has a multiplicity stating how many data values an instance of the class should hold, which is absent in Connection. (We ignore some other properties of the Attribute construct, but these can be added similarly.) Therefore, a new class is defined in Boom, called MultiplicityConnection. This class is a subclass of Connection; it adds an extra attribute, *multiplicity*, and a set of corresponding methods. Furthermore, the class also defines the extra semantics implied by the multiplicity property.

The operations of a UML class are defined similarly to attributes using (subclasses of) the class Operation in Boom.

The specification of the Data Type construct can also be done with the Declaration class, because the construct supports the same features as the Class construct. However, instances of these two constructs have different semantics. They are therefore specified using two different BOOM classes: Object is an ordinary kind of instance, while a Data Value cannot modify its own state. Therefore, the Object construct is specified using the Instance class, while Data Value is specified using the TokenInstance subclass of Instance.

In UML the connection between an object and a data value is called an Attribute Link. It corresponds to an attribute in the object's class, and it holds the actual attribute values. However, the number of data values referenced by the attribute link must fulfil the requirement stated by the multiplicity declared by the attribute. This is specified in the NumberedLink class in BOOM.

Hence, we state how these constructs are mapped onto BOOM and how they may be connected with the following ODAL expressions:

```
mapping ADD (EntityType NEW (CLASS, Declaration));
mapping ADD (EntityType NEW (DATATYPE, Declaration));
mapping ADD (EntityType NEW (OBJECT, Instance));
mapping ADD (EntityType NEW (DATAVALUE, TokenInstance));
mapping ADD (ConnectionType NEW (ATTRIBUTE, MultiplicityConnection));
mapping ADD (LinkType NEW (ATTRIBUTELINK, NumberedLink));
ruleset ADD (AssociationRule NEW (CLASS, DATATYPE, ATTRIBUTE));
```

The meta-model defining these constructs and how they are related to each other is found in Figure 3.

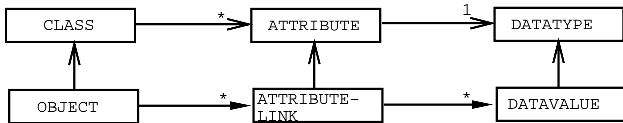

Fig. 3. A fragment of a meta-model presenting the Class and Attribute constructs of UML.

Note that this kind of diagram only present the constructs and how they are related. The detailed formal semantics of each construct is specified with the ODAL language, and is found in the BOOM classes that each construct maps onto.

3.2 Association Construct

The Association construct of UML is a relationship with at least two end-points. These end-points are similar to the attributes of a class; they have a name and a multiplicity. (Once again we ignore some of its properties, but these can be specified using the same technique as in the case of *multiplicity*.) We therefore define the Association End construct to be specified by the MultiplicityConnection class. This class has an extra attribute representing the multiplicity, as well as some methods using this attribute. The Association construct itself can be

specified with the Declaration construct, since in UML an Association can create instances, called Links, and it has a name and includes operations for reading and modifying the end-points of these Links. There is, however, an extra requirement that must be fulfilled by an Association which is absent in a Class: it must have at least two association end-points (see Figure 4). Therefore, a new class is defined in Boom, called AssociationDeclaration. It is a subclass of Declaration; the difference is that the subclass has an extended invariant stating the extra requirement. (In fact, there are also other differences between the N-ary Association construct and the Class construct, and all these differences are captured by the AssociationDeclaration class in Boom.)

To specify the UML Association construct the following Odal expression establish the mapping between the constructs:

```
mapping ADD (EntityType NEW (ASSOCIATION, AssociationDeclaration));
mapping ADD (ConnectionType NEW (ASSOCIATIONEND, MultiplicityConnection));
ruleset ADD (AssociationRule NEW (ASSOCIATION, CLASS, ASSOCIATIONEND));
```

Similarly, the Link and Link End constructs of UML correspond to the Object and Attribute Link constructs. However, Link End has an additional requirement: it must be connected to exactly one instance, and not to a set of instances. Hence, a new Boom class is defined: SingleLink.
These two constucts are mapped onto Boom with the following expressions:

```
mapping ADD (EntityType NEW (LINK, Instance));
mapping ADD (LinkType NEW (LINKEND, SingleLink));
```

The meta-model specifying these constructs and how they are related to each other is presented in Figure 4.

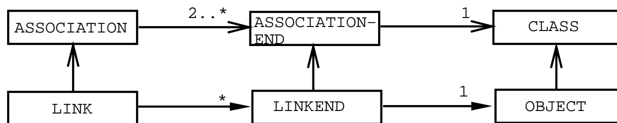

Fig. 4. A fragment of a meta-model presenting the Association and Link constructs of UML.

Once again, the details of the specifications are to be found in the Boom classes. The diagram only presents the constructs and their relationships. A formal specification in Odal of some of the relationship constructs in UML is found in [11].

3.3 Package Construct

In UML Packages are used for grouping elements, such as Classes and Packages, into units that act as name spaces for their contained elements, i.e. each contained element must have a unique name within the package. Furthermore, a

package is not instantiable, which implies that it does not exist at the object level; its sole purpose from a semantic point of view is to define a name space. We will use the Package construct of UML to exemplify how relationships at the model level are expressed in BOOM by specifying that packages can contain classes.

In BOOM the specific connection between two instances is expressed with a Link, while the declaration of that connection is expressed with a (subclass of) Connection. This implies that the relationship between a specific package and its contents is expressed with a Link. Hence, the declaration of this Link is expressed between the creator of the package and the creator of the contents. We therefore introduce the MetaClass and the MetaPackage constructs. When these constructs are instantiated, Classes and Packages are created. Hence, the Contents relationship, stating that a package may contain classes, is declared from MetaPackage to MetaClass. In this way each package has a link expressing the connection between the package and its contained classes.

To specify the MetaClass construct we use the MetaDeclaration class of BOOM, which is a subclass of Declaration, but which creates Declarations instead of Instances. The MetaPackage construct might also have been specified with the MetaDeclaration class, but the construct has an additional constraint: it must have a containment relationship (see Figure 5). Therefore, the MetaPackage construct is mapped onto the NamespaceDeclaration, which is a subclass of MetaDeclaration.

```
mapping ADD (EntityType NEW (METAPACKAGE, NamespaceDeclaration));
mapping ADD (EntityType NEW (METACLASS, MetaDeclaration));
```

The Contents construct is specified with a subclass of Connection, because its semantics affects the uniqueness of the contained classes' names as well as the semantics of the element-lookup operation. The actual connection between a package and its contained classes is defined with a Link.

```
mapping ADD (ConnectionType NEW (CONTENTSDECLARATION, ContentsConnection));
mapping ADD (LinkType NEW (CONTENTS, Link));
ruleset ADD (AssociationRule NEW (METAPACKAGE, METACLASS, CONTENTSDECLARATION));
```

As packages are not instantiable, the Package construct is mapped onto the NonInstantiableDeclaration, which is a subclass of Declaration. The difference between the two is that the subclass cannot create instances.

```
mapping ADD (EntityType NEW (PACKAGE, NonInstantiableDeclaration));
```

The meta-model specifying the contents relationship between the Package construct and the Class construct is presented in Figure 5.

3.4 Class and Association Constructs Revisited

In this section we give two examples showing why the BOOM framework supports adjustments of the modelling language. In UML, not only packages but also classes may contain other classes. This implies that the container class acts as the

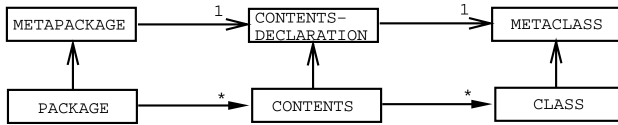

Fig. 5. A fragment of a meta-model presenting the Package and Contents constructs of UML.

name space of the contained classes. Since we already have specified the meaning of the containment relationship, and thanks to the localization principle, it is very simple to add this extension to the current specification. We only need to do two modifications of the specification: i) the set of association rules is extended with the possibility for a class to have a containment relationship to other classes, and ii) the mapping of the MetaClass construct onto the Boom class which specifies its semantics is changed. Instead of mapping it onto MetaDeclaration it is mapped onto NamespaceDeclaration, which has the same semantics as a MetaDeclaration, but with the additional invariant constraint of always including a containment relationship. No other modification is needed in the specification. The meta-model is updated accordingly (see Figure 6).

```
mapping REPLACE (EntityType NEW (METACLASS, NamespaceDeclaration));
ruleset ADD (AssociationRule NEW (METACLASS, METACLASS, CONTENTSDECLARATION));
```

The second example is the Association Class construct. In UML, an association may have attributes attached to it. These attributes model information that belongs to the relationship itself and not to any of the associated classes. This kind of association is called an Association Class for it acts both as an association and as a class. To specify the Association Class construct in Boom, we use the same Boom classes as for Association and Link, but we add to the specification rules that makes it possible for the construct to contain links and attribute links.

```
mapping ADD (EntityType NEW (ASSOCIATIONCLASS, AssociationDeclaration));
mapping ADD (EntityType NEW (LINKOBJECT, Instance));
ruleset ADD (AssociationRule NEW (ASSOCIATIONCLASS, DATATYPE, ATTRIBUTE));
ruleset ADD (AssociationRule NEW (ASSOCIATIONCLASS, CLASS, ASSOCIATIONEND));
```

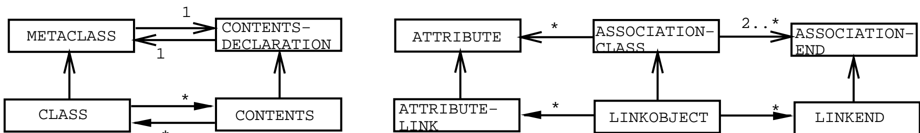

Fig. 6. A fragment of a meta-model presenting the additions made to the model: Class containment and Association Class.

This example shows how simple it is to add a new construct to the language if the necessary Boom classes are available. Note that already existing classes can be reused when defining new classes.

3.5 Other Meta-modelling Approaches

A few other approaches to meta-modelling of modelling languages have been presented:

MOF The *Meta Object Facility* (MOF) [9] is a meta-meta model, i.e. a model for modelling meta-models. The MOF has been accepted as a standard by the Object Management Group (OMG), and it is, for example, used as the foundation for defining UML. The main purpose of the MOF is "to provide a set of CORBA [Common Object Request Broker Architecture [14]] interfaces that can be used to define and manipulate a set of interoperable metamodels" [9, 1-1]. This implies that the MOF can be used for defining repository services for storing computer representations of models, for information management as well as for data warehouse management. Hence, the MOF is a model for meta-data management and meta-data interoperability. The focus is on tool vendors and tool interoperability. However, the MOF is not intended for specification of the semantics of a meta-model. The meaning of the different constructs in a meta-model has to be expressed by other means.

CDIF The *CASE Data Interchange Format* [3] is a standard for interchange formats between CASE tools. It is not specific for object-oriented models, and can be used for several different kinds of models, like process models and data-flow models. The models in CDIF are defined using *Entity-Relationship-Attribute* models. These models are defined at three levels of models: model, meta-model and meta-meta model. No run-time semantics or modelling semantics is given as the focus is on interchange of information between CASE tools.

4 Concluding Remarks

In this paper we have presented Boom, a framework for formal specification of modelling languages. The framework consists of a formal specification language and a meta-model of object-oriented modelling languages. The meta-model offers a set of predefined components to be used when specifying such a modelling language. Hence, the specification work does not have to start from scratch each time, but can be based an already existing specification. Differences between the actual semantics and the semantics offered by the components are specified in user-defined subclasses of the components.

The framework offers different levels of abstractions for the specification of a language: what constructs exist, how the constructs are interconnected, a mapping from the constructs to Boom classes, and the detailed semantics defined in

the Boom classes. In this way, all the details need not be covered for understanding the specification of a language. This paper focuses on the first three layers; a description of the last layer can be found in e.g. [13, 11].

The practical usage of the framework is demonstrated by the specification of the Unified Modeling Language, and this paper presents how some of the constructs in UML have been specified. This paper also demonstrates that many typical modifications of a modelling language, like adding variations to existing constructs, or modifying the detailed semantics of a construct, can be easily performed in the framework. In our work we have also used Boom for specifying other parts of UML, e.g. specification of the abstract syntax, the static and the dynamic semantics of the Collaboration, the Use Case and the Subsystem constructs.

There are three major areas that will benefit from the Boom framework, all of them requiring formal and adjustable specification techniques. Use of the Boom framework will facilitate tailoring of the modelling language in a development process: the Boom classes define the abstract syntax and semantics of language constructs, such as Class, Association, and Object. The definitions of the constructs used in a project must capture the desired semantics to ensure that the constructs are used consistently within the project. Moreover, the framework facilitates the development of tools for performing intelligent operations on models, such as checking for internal consistency or conformance between parts of the model. The Boom classes can be used to define precisely what such operations mean. Finally, we claim that Boom is of particular use for the UML development community, where the absence of a proper and flexible semantics makes it difficult to relate the many different approaches and extensions.

References

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

[2] E.H. Dürr and J. van Katwijk. VDM++ - A Formal Specification Language for Object-oriented Designs. In *Computer Systems and Software Engineering. Proceedings of CompEuro'92*, pages 214–219. IEEE Computer Society Press, 1992.

[3] Electronic Industries Association, 2500 Wilson Blvd. Arlington, VA 22201. *EIA/IS-107: CDIF / Framework for Modeling and Extensibility*, 1997. http://www.eia.org/eig/cdif/how-to-obtain-standards.html.

[4] International Telecommunication Union (ITU), Place des Nations, CH-1211 Geneva 20, Switzerland. *Recommendation Z.100 (03/93) - CCITT specification and description language (SDL)*, 1993. http://www.itu.int/itudoc/itu-t/rec/z.html.

[5] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process.* Addison-Wesley, 1999.

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

[7] 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.

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

[9] *Meta Object Facility (MOF) Specification*, September 1997. On-line documentation: http://www.omg.org/pub/docs/ad/97-08-{14,15}.pdf.

[10] Object Management Group, Framingham Corporate Center, 492 Old Connecticut Path, Framingham MA 01701-4568. *OMG Unified Modeling Language Specification, version 1.3*, June 1999. http://www.omg.org/cgi-bin/doc?ad/99-06-08.

[11] G. Övergaard. A Formal Approach to Relationships in the Unified Modeling Language. In M. Broy, D. Coleman, T. S. E. Maibaum, and B. Rumpe, editors, *Proceedings PSMT'98 Workshop on Precise Semantics for Software Modeling Techniques*, pages 91–108. Technische Universität, München, Germany, TUM-I9803, April 1998.

[12] G. Övergaard. A Formal Approach to Collaborations in the Unified Modeling Language. In R. France and B. Rumpe, editors, *Proceedings of UML'99 – The Unified Modeling Language: Beyond the Standard, Lecture Notes in Computer Science 1723*, pages 99–115. Springer-Verlag, 1999.

[13] G. Övergaard and K. Palmkvist. A Formal Approach to Use cases and Their Relationships. In P.-A. Muller and J. Bézivin, editors, *Proceedings of the Unified Modeling Language: UML'98: Beyond the Notation, Lecture Notes in Computer Science 1618*. Springer-Verlag, 1999.

[14] A. Pope. *The Corba Reference Guide : Understanding the Common Object Request Broker Architecture*. Addison-Wesley, 1998.

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

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

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

[18] S. Stepney, R. Barden, and D. Cooper. *Object Orientation in Z*. Springer-Verlag, 1992.

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