A Rigorous Metamodel for UML Static Conceptual Modelling of Information Systems

Régine Laleau[1] and Fiona Polack[2]

[1] CEDRIC-IIE(CNAM),
18 allée Jean Rostand, 91025 Evry, France
laleau@iie.cnam.fr
[2] Department of Computer Science
University of York, York, YO10 5DD, UK
fiona@cs.york.ac.uk

Abstract. Object-oriented specification and design approaches, such as the UML, are used in many sectors, including information systems development. One reason for the popularity of the UML is that it has notations for many types of system and all stages of development. However, this also makes it cumbersome and semantically imprecise. This paper looks at a UML for information systems specification. It both selects from and extends the UML1.3, defining the semantics of the IS UML in B-style invariants. The paper discusses the relationship of the work both to the current UML metamodels, and to proposals for extensions to the UML.

1 Introduction

The Unified Modeling Language (UML) is still evolving, but is widely used in commercial software engineering. The Object Management Group (OMG), which oversees the development of the UML has recognised the need to allow specialisation of the language for different types of system. The mechanisms of extension are not static or agreed, and Requests for Proposals (RFPs) have been issued, prior to evolution of a UML version 2.0.

The UML metamodels have varied over the years, but have all been general-purpose. The language contains concepts for abstract and concrete expression, and for many types of system (including, for example, real time and concurrent systems). One problem for users of UML is to determine the subset of notations which is relevant to their particular project, or to their particular level of abstraction.

This paper presents a variant of UML1.3 suitable for the specification of the static structure of an information system (IS). This derives from ongoing work on IS specification at CEDRIC-IIE, where an approach has been developed to derive formal specifications (in the B notation[1]) from object-oriented class and behavioural models[21, 20, 13].[1]

[1] Metamodels for relevant behavioural notations have been defined, as well as the static model. Work on behavioural semantics is in progress.

1.1 Metamodel Presentation: the B Method

The IS UML semantics are presented as metamodels, which are compared to the OMG's UML 1.3 semantics (http://www.rational.com/uml). The presentation uses UML class diagrams, for the basic structure. The structure, detail and constraints are then expressed mathematically, as B invariant clauses.

B notations are used for a number of reasons:

- B underpins the IS specification work, on which this paper is based.
- B provides the needed power of expression. It includes standard operators on sets, functions and relations, and types such as sequences, allowing a precise, expression of invariants.
- A formal notation such as B supports reasoning, such as the verification of consistency of the metamodels. Meta-rules can also be defined to assist in the execution of proofs about the specification.
- A formal metamodel would assist development of advanced modelling tools. The formal meaning can be built in, such that diagrammatic models of a system are formally correct.

The B invariants use only notations in common usage in set theory and predicate logic. The intuitive meaning of each invariant is given in the text. This paper first considers the bases for modification of the UML. It then illustrates the metamodel for static structure, the rationale behind it and its relationship to UML1.3.

2 UML Specialisation and Extension

The UML1.3 variant illustrated here does not conform to the OMG's current UML extension mechanisms, because the UML mechanisms do not provide the relevant facilities[2, 9, 4].

The OMG structures its metamodel for UML1.3 using *packages*. However, an OMG package has limited power: it is not allowed to import another package in order to elaborate partially-defined features from that package[6, section 2.1.2.8]. Since all the features of a concept must be stated in one package, it is not possible to separate out the UML specification features, or the components relevant to IS development using packages.

The OMG *profile* was proposed as a more flexible extension mechanism. The definition recommended for UML2.0 RFPs (see [9]) defines the profile as a subset of the UML metamodel, with additional well-formedness rules (in OMG's object constraint language, OCL). All profile concepts must be expressible in terms of existing UML metamodel concepts or stereotypes extending these concepts. An OMG profile cannot add structure (associations) or entirely-new concepts to the metamodel. Although the IS metamodel can express many of the required features in terms of existing UML1.3 concepts, it adds to and modifies the structure of the metamodel. It also modifies some semantics.

OMG also has a more robust extension approach, using the MetaObject Facility (MOF), but this is complicated by the fact that the UML metamodels

are not MOF-compliant, and no rules are given for the syntactic or semantic interpretation of UML concepts under extension[9].

A number of alternative extension mechanisms have been proposed, motivated by a desire to shrink and focus the core UML[12, 4, 10]. These are more appropriate to this work. We intend to move towards an IS UML following the MOF-compliant pUML preface definition[6, 9]. The pUML, like OMG, uses OCL to complement the diagrammatic metamodels[2]. However, OCL does not have a precise semantics; at this stage, we prefer to use a formal language with well-defined semantics for describing specialisation of and extensions to the UML notations.

3 Simplifying the UML for IS Specification

An IS manages large amounts of highly structured data. A static model is essential, along with a large number of data constraints defining the integrity conditions of the IS. In a well-designed IS, operation preconditions maintain integrity by enforcing the data constraints. (This is explored formally in [15, 17] and used in work on the formal derivation of databases[18]. Tool support is described in [19].)

This paper focuses on the semantics of class models for use in IS specification. Features of the general OMG metamodels which are rarely relevant to IS or to specification are omitted. These include attributes of metamodel classes concerning concurrency, real-time operation, visibility and scope. Relevant extracts of the OMG's UML1.3 metamodel are given in the appendix to this paper.

Complex subclass hierarchies, used in the OMG metamodels to handle general system features, are collapsed and simplified. For example, the *Name* subclass in the *ModelElement* hierarchy of the UML1.3 fundamentals is demoted to an attribute of classes representing named constructs.

The simplification of the UML1.3 structures facilitates expression of some semantic features such as association qualifiers, and improves the readability of the metamodel diagrams. However, the collapsing of the OMG structures sacrifices some classification of semantic elements.

The IS metamodel (figure 1) brings together all relevant static UML features except n-ary associations and aggregation/composition. A n-ary association can always be replaced by binary associations and constraints. Aggregation associations imply constraints on the lifecycles of objects and the operations of the classes concerned. Work is in progress on this[24]. At this stage, OMG attributes recording aggregation/composition (and those relating to navigability) have been omitted from the metamodels.

Some structure is omitted for simplicity. For example, in the OMG metamodels, all generalisable features are subclasses of *GeneralizableElement* (appendix,

[2] The formalisation of diagrammatic syntax and semantics dates from research tools for the formal review of diagrammatic specifications[25]. Initial UML work represented the whole UML metamodel in a formal notation, such as Z[8, 5]. pUML now uses UML diagrams extended with OCL constraints [3, 7].

figure 5). For the IS class model, we restrict generalisation to classes, because the semantics of other generalisations (eg association) is not well defined.

Metamodel associations such as those between *Class* and *Attribute*, and between *Class* and *Operation*, are derived from the OMG metamodels by combining references to the various classes and then collapsing the hierarchies (appendix, figure 5). The OMG's elaborate modelling of association class semantics, using multiple inheritance (appendix, figure 6), is replaced by a simple association.

The OMG metamodel coverage of integrity constraints[16, p2-14] is not adequate for IS. This is addressed in section 6, below.

4 Diagrammatic Metamodel for the IS Class Models

A class diagram provides the basic metamodel structure for the IS UML (figure 1). The discussion focuses on the meaning (the expressed semantics) and on the relationship between this metamodel and those of the OMG. The metamodel can be thought of as extending rather than replacing semantics of the OMG metamodels.

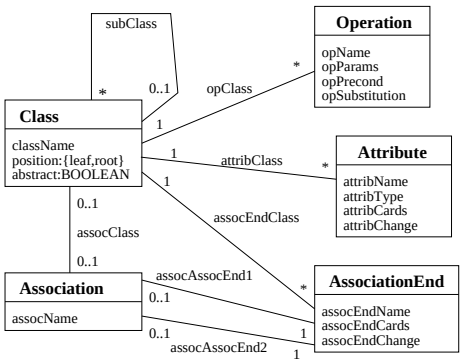

Fig. 1. Metamodel for the IS Class Model

The attributes of the IS *Class* are implicitly inherited from the OMG's *GeneralizableElement* and *ModelElement*. Inheritance is expressed by the association *subClass*, a self-reference on *Class*. Any instance of *Class* is thus linked to its subclasses and superclass. For simplicity, multiple inheritance is not considered here.

The classes *Attribute* and *Operation* express the structure and behaviour of a class. A class may have many attributes and many operations, but each attribute or operation is directly linked to one class. Subclasses inherit the characteristics of their superclasses via the *subClass* association.

Operation has a name, a signature, a precondition and a substitution. It does not include attributes for concurrency, since this is conventionally handled in IS by a database management system.

The attributes of *Attribute* are from the OMG superclasses, omitting those relating to visibility and scope.

Association and *AssociationEnd* have name attributes (implicit inheritance from the OMG *ModelElement*). *AssociationClass* is modelled by the association *assocClass* between *Class* and *Association*. The semantics are elaborated below.

5 Formal Semantics: Extending the Diagrammatic Metamodels

The B invariants expressing the semantics of the IS UML class models can be typechecked (and, if required, used for further rigorous development, including refinement and proof).

In order to exploit the B tool support, the whole model, including the elements already expressed in diagrams, needs to be expressed in B. This section illustrates the precise expression of the semantics. The formal invariants representing the metamodel structure are derived using existing guidelines for representing object-oriented class diagrams in B[21, 14]. The formal metamodel is extended with invariant clauses to clarify constraints among the components. The instantiation of classes to objects and associations to links is used to express multiplicity and other constraints.

This paper does not present the full definitions. The subsections illustrate typing information, the data structure, and finally the additional semantics required to express the full meta-language.

5.1 Type Structures for the Formal Semantics

Data types (written in upper case) can be equated to mathematical sets. The B Method incorporates integers (and integer-related proof rules), and has a variety of mechanisms for introducing new sets, at various levels of detail. In this presentation, most types are defined as given sets, about which most details remain unelaborated. A B given set may have enumerated values, or may simply be an (implicitly) non-empty, finite set[1, sections 2.2 and 4.17]. A given set called *TYPE* is defined here to represent UML- and user-defined data types.

The type of each class in the diagrammatic metamodel is a given set, as is that of the class *Object*, used to express multiplicity semantics. An invariant clause (table 1) introduces sets of known instances of each class as a subset of the class type.

5.2 Structural Definitions Using B Relations and Functions

A binary relation is a set of ordered pairs, linking an element of the source set (the domain, dom) to an element of the target set (the range, ran)[1, p77-9].

Table 1. B Invariants for Sets of Known Instances of Metamodel Classes

$class$	$\subseteq$	$CLASS$
$association$	$\subseteq$	$ASSOCIATION$
$attribute$	$\subseteq$	$ATTRIBUTE$
$operation$	$\subseteq$	$OPERATION$
$object$	$\subseteq$	$OBJECT$
$associationEnd$	$\subseteq$	$ASSOCIATIONEND$

The inverse of a relation ($relation^{-1}$) defines the reverse ordered pairs (ie. range first). The relation is a mathematical model of a many-to-many association.

Participation of domain and range elements can be constrained to mathematical functions[1, p86]. These can model any degree of association (table 2).

Table 2. Example Mappings Modelled by Relation and Functions

$\leftrightarrow$	*relation*	$*$ *to* $*$ *mapping*
$\rightarrow$	*function*	$*$ *to* 1 *mapping, all domain takes part*
$\rightarrowtail$	*injection*	1 *to* 1 *mapping, all domain takes part*
$\nrightarrow$	*partial function*	$*$ *to* 1 *mapping, not all domain takes part*
$\nrightarrow\!\!\!\!\!\rightarrowtail$	*partial injection*	1 *to* 1 *mapping, not all domain takes part*
$\twoheadrightarrow$	*partial surjection*	$*$ *to* 1 *mapping, not all domain, all range takes part*

Functions are used to model the associations in the metamodel (table 3). Functions are also used to associate attributes to classes (table 4). A total function ($\rightarrow$) from the set of known instances of the class to the type of an attribute models a mandatory, mono-valued attribute. Injections ($\rightarrowtail$) are used for mandatory attributes with a unique value. Partial functions ($\nrightarrow$) define optional, mono-valued attributes. There are no multi-valued attributes in the metamodel.

Most of the attribute types are OMG types[16, section 2.7]. *NAME* is an implicit subset of the OMG's catch-all *STRING* type. The ranges of *attribCards* and *assocEndCards* are sets of natural numbers drawn from the powerset, $\mathbb{P}\,\mathbb{N}$. This allows any cardinality value to be expressed, with B's upper limit (*maxint*) equating to UML's *. The range of *opParams* is a sequence $seq(TPARAM)$. This is an indexed structure (mathematically, it is an injection from natural numbers to the sequence type), and is used here to model an ordered list of parameters. The type *TPARAM* is a triple of a name, a basic UML type, and an indicator of whether the parameter is input (i) or output (o). (This simplifies the OMG representation, which additionally has *inout* and *return* parameters.)

Table 3. B Invariants Representing Associations in the IS Metamodel

$subclass$	$\in$	$class \nrightarrow class$
$assocClass$	$\in$	$association \rightarrowtail class$
$assocAssocEnd1$	$\in$	$association \rightarrowtail associationEnd$
$opClass$	$\in$	$operation \rightarrow class$
$assocAssocEnd2$	$\in$	$association \rightarrowtail associationEnd$
$attribClass$	$\in$	$attribute \rightarrow class$
$assocEndClass$	$\in$	$assocEnd \rightarrow class$

$$TPARAM \in (NAME \times TYPE \times \{i, o\}) \ .$$

The range of *opPrecond* is the OMG type, *PREDICATE*. This type is not elaborated in UML1.3. However, the semantics could be defined more fully. For example, a predicate is only valid if it evaluates to a boolean (true or false). A semantically-relevant predicate should not always evaluate to the same value.

Two of the constraining invariants on the structure modelled in figure 1 express the fact that attributes (operations) of a class comprise those declared in that class and in all of its superclasses. For a given class, c:

$$\begin{aligned} allClassAttributes[\{c\}] &= \bigcup\{attribClass^{-1}[\{c'\}] \mid c' \in subclass^*(c)\} \\ allClassOperations[\{c\}] &= \bigcup\{opClass^{-1}[\{c'\}] \mid c' \in subclass^*(c)\} \ . \end{aligned}$$

The notation subclass*, reflexive transitive closure, models the whole subclass hierarchy including the root class. A hierarchy without the root class is modelled by $subclass^+$, transitive closure. This is used in the invariant which prevents cycles of inheritance,

$$subclass^+ \cap id(class) = \varnothing \ .$$

5.3 Semantics of Class Models Under Instantiation

The semantics of instantiated class models express, for instance, cardinalities and multiplicities. In the OMG metamodels, these are expressed diagrammatically [16, figure 2-16]; here they are modelled in B invariants.

The instantiation of a class is an object (table 1), and each object is one instantiation of exactly one class:

$$objectClass \ : \ object \rightarrow class \ .$$

All the objects of a class, taking into account inheritance links, is defined as,

$$allClassObject = (subclass^*)^{-1} \fatsemi objectClass^{-1} \ .$$

The composition operator, $\fatsemi$, can be thought of as the navigation of the metamodel associations.

Table 4. B Invariants Associating Attributes to IS Metamodel Classes

CLASS	*ATTRIBUTE INVARIANT*
Class	$className \in class \rightarrowtail NAME$
	$position \in class \rightarrow \{leaf,\ root\}$
	$abstract \in class \rightarrow BOOLEAN$
Attribute	$attribName \in attribute \rightarrow NAME$
	$attribType \in attribute \rightarrow TYPE$
	$attribCards \in attribute \rightarrow \mathbb{P}\ \mathbb{N}$
	$attribChange \in attribute \rightarrow BOOLEAN$
Operation	$opName \in operation \rightarrow NAME$
	$opPrecond \in operation \rightarrow PREDICATE$
	$opSubstitution \in operation \rightarrow STRING$
	$opParams \in operation \rightarrow seq(TPARAM)$
Association	$assocName \in association \nrightarrow NAME$
AssociationEnd	$assocEndName \in associationEnd \nrightarrow NAME$
	$assocEndCards \in associationEnd \rightarrow \mathbb{P}\ \mathbb{N}$
	$assocEndChange \in associationEnd \rightarrow BOOLEAN$

The definition of *object* is used to instantiate associations as sets of links among objects. There may be more than one association among the same objects:

$$linkAssociation\ :\ association\ \rightarrow\ (object\ \leftrightarrow\ object)\ .$$

The semantics of association ends requires that the ends of an association are different. This is stated as a partition of the set of association ends:

$$\begin{array}{l} \text{ran}\ (assocAssocEnd1)\ \cup\ \ \text{ran}\ (assocAssocEnd2)\ = assocationEnd\ \wedge \\ \text{ran}\ (assocAssocEnd1)\ \cap\ \ \text{ran}\ (assocAssocEnd2)\ =\ \varnothing\ . \end{array}$$

An additional invariant is required to constrain participation in the instantiation to objects of the classes implied by the association:

$$\begin{array}{l} linkAssociation\ \subseteq \\ ((assocAssocEnd1\ ;\ assocEndClass\ ;\ allClassObject)\ \otimes \\ \ (assocAssocEnd2\ ;\ assocEndClass\ ;\ allClassObject))\ . \end{array}$$

This expression states that the links which instantiate an association are drawn from all pairs (constructed using the direct product, $\otimes$) of objects such that the first element comes from *assocAssocEnd1* and the second from *assocAssocEnd2*. The objects are those which instantiate the class (with all its superclasses) for the relevant association end.

An object which is the instantiation of an association class corresponds to one and only one link of the relevant association. The *card* operator returns the number of elements of a set, relation or function:

$$\forall\, a \in \mathrm{dom}\ assocClass \bullet$$
$$card(linkAssociation(a)) = card(allClassObject(assocClass(a))) .$$

This expression states that, for any association which has an association class, the number of instantiations of that association must be the same as the number of instantiations of the association class. This is more logical than the OMG semantics (using multiple inheritance): an association class is a simple class, which assists in the expression of any constraints, associations and inheritance structures in which it participates.

The multiplicity of association ends is expressed as a general constraint that the number of instantiated links must be a permitted cardinality of the association end:

$$\forall\, o \in object \bullet$$
$$card((linkAssociation(assocAssocEnd1(ae))$$
$$\cup\ (linkAssociation(assocAssocEnd2(ae))^{-1}))[\{o\}])$$
$$\in assocEndCards(ae) .$$

The expression refers to a specific association end, *ae*. It extracts and instantiates the association of which *ae* is the association end numbered 1. This gives a set of pairs of objects with those at the *ae* end first. The next part of the expression extracts and instantiates the other end of the association, giving a set of pairs of objects with those at the *ae* end second. To get the total number of objects taking part in the association at the *ae* end, the second pair is inverted (so that those at the *ae* end are first). The two sets of pairs are unified. The objects linked to each object of the class at the *ae* end are extracted (using a relational image of *o*, $[\{o\}]$) and counted using the *card* operator. The number of objects must be one of those defined in the set of numbers which is the value of the *assocEndCards* attribute of the association end, *ae*.

Instantiation is used to elaborate the semantics of attribute values and attribute cardinality. The instantiation of an attribute uses the given set, *VAL*, representing all possible values in the model. An object maps to many values, representing the instantiations of its attributes:

$$attribObjectValue : attribute \rightarrow (object \leftrightarrow VAL) .$$

When instantiated, an object can take only values for the attributes of its class (including superclasses):

$$\forall\, c \in class \bullet$$
$$\mathrm{dom}(\bigcup(attribObjectValue\ [allClassAttributes[\{c\}]])) \subseteq objectClass^{-1}[\{c\}] .$$

When any object is instantiated, the number of values assigned to each attribute *a* (including inherited attributes) must be a valid cardinality for that attribute:

$$\forall\, o \in object \bullet$$
$$\forall\, a \in (allClassAttributes(objectClass(o))) \bullet$$
$$card(attribObjectValues(a)[\{o\}]) \in attribCards(a) .$$

6 Static Integrity Constraints

In IS, complex relationships and restrictions on data are expressed as constraints. In general, these constraints are available to the developer when writing operations on the IS. In particular, cardinality, key and inter-association constraints can be used to automatically generate operation preconditions[15, 17]. The OMG metamodels do not provide sufficient concepts to express the complexity and variety of IS constraints[16, p2-14]. The IS metamodel, of which figure 2 is an extract, is sufficient to express both general IS integrity concepts and data constraints specific to a particular IS. The structure specialises the OMG class *StaticConstraint*, and uses the IS metamodel classes *Class*, *Attribute* and *Association*.

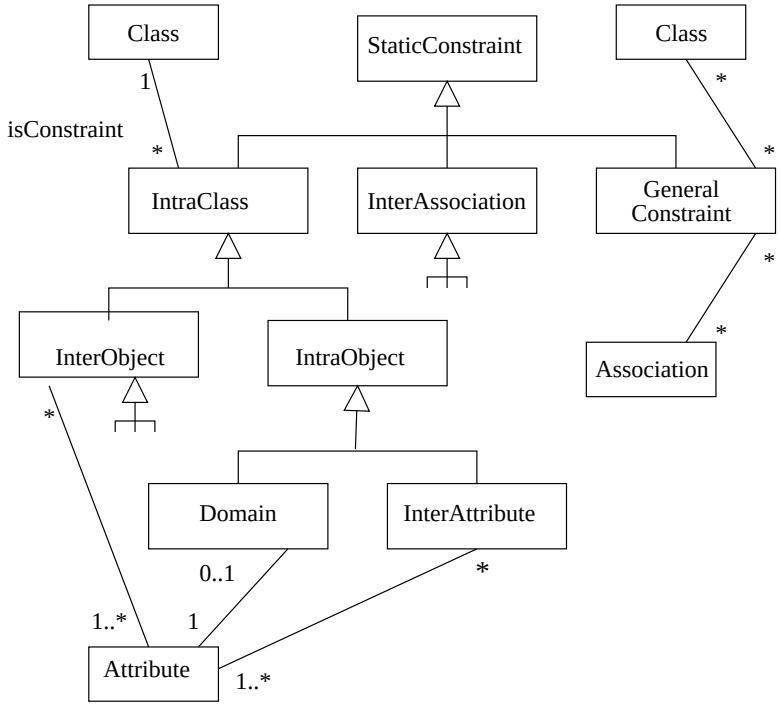

Fig. 2. Metamodel Extract for the IS Static Constraints

Of the three top-level subclasses in figure 2, *GeneralConstraint* provides the structure for data constraints involving any combination of classes, attributes and associations. *InterAssociation* is considered below (section 6.2). *IntraClass* defines constraints within one class. The subclass *IntraObject* can express con-

straints on domains and among attributes of one object. *InterObject* is considered below (section 6.1).

6.1 *InterObject* : Keys and Dependency Constraints

It is common practice in IS specification to identify candidate keys, sets of attributes from a class which can uniquely identify any instance, and upon which other attributes are functionally dependent. To express these concepts, the IS metamodel in figure 1 is extended (figure 3) with *Key* and *FunctionalDependency*, two subclasses of *InterObject*. The association *isKey* is a specific link of the *isConstraint* association in figure 2

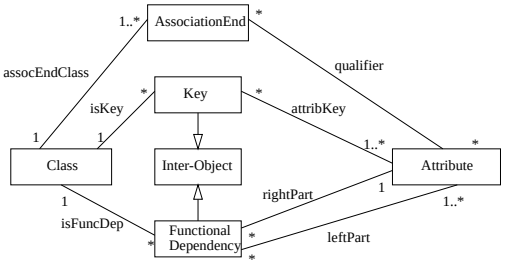

Fig. 3. Metamodel Elaborating Key and Dependency Constraints

Invariants again capture and elaborate the semantics. The association *isKey* is a function from *key* to *class*, whilst *attribKey* is represented as a relation between *key* and *attribute*. The additional constraint relates the attributes in the range of *attribKey* to the class in the range of *isKey* via the *key*.

$$
\begin{aligned}
& isKey \in key \rightarrow class \\
\wedge\ & attribKey \in key \leftrightarrow attribute \\
\wedge\ & \mathrm{dom}\ (attribKey) = key\ .
\end{aligned}
$$

The attributes of a key must be attributes of its class or superclasses:

$$\forall\, c \in \mathrm{ran}\ (isKey) \bullet attribKey[isKey^{-1}\ [\{c\}] \subseteq allClassAttributes[\{c\}]\ .$$

The uniqueness of keys requires that there is at most one object for each combination of values of the key of a class, and that all attributes in the key have values (the cardinality of the attribute cannot be 0). The constraints are defined on the set of attributes *attribKey[{K}]*, forming the key K of a class *cl*, and uses the fact that if the key value for two objects (*o1* and *o2*) are the same, then the two objects are in fact one object:

$$\forall K \in key \bullet 0 \notin attribCards(attribKey[\{K\}])$$
$$\wedge\ isKey(K) = cl \Rightarrow (\forall o1, o2 \in allClassObject(cl) \bullet$$
$$\forall a \in attribKey[\{K\}] \bullet$$
$$(attribObjectValue(a)(o1) = attribObjectValue(a)(o2))$$
$$\Rightarrow (o1 = o2)) .$$

The B invariants on qualifier and functional dependency are not illustrated here. The semantics of the qualifier are that an association can be qualified by a subset of attributes of the opposing class. Functional dependency semantics define that the attributes participating in the *leftPart* determine the value of those in the *rightPart*. These are disjoint sets of attributes.

6.2 *InterAssociation* : Participation Constraints

Inter-association constraints express constraints on participation (set exclusion, totality, subset)[22] among associations or association ends. (An example of such a constraint is that expressed above, stating that the ends of an association are different.)

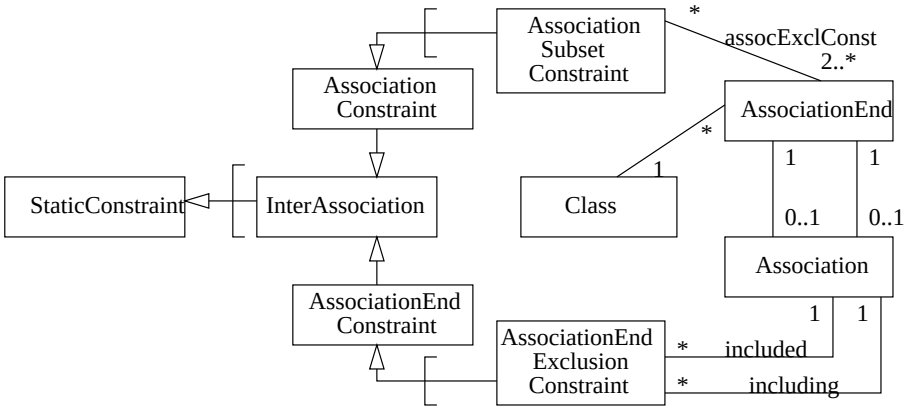

Fig. 4. Metamodel Elaborating Inter-association Constraint Semantics

Figure 4 extends figures 1 and 2, above. *InterAssociation* is specialised into *AssociationConstraint* and *AssociationEndConstraint*. One illustrative subclass is shown for each of these.

AssociationSubsetConstraint allows expression of the fact that links of one association may be a subset of links for another association. In metamodel terms, this is defined by the associations (*included* and *including*) between the constraint class and the *Association* class.

For association ends, the illustrated subclass is an exclusion constraint. This constraint is stated among two or more instances of *AssociationEnd*. Two invariants are shown.

If an exclusion constraint *ec* is defined between the association ends, the ends must be linked to the same class:

$$\forall\, ec \in AssociationEndExclusionConstraint \bullet$$
$$card(assocEndClass \fatsemi assocExclConst[\{ec\}]) = 1 .$$

If two association ends *ae1* and *ae2* participate in an exclusion constraint *ec*, then an object of the common class must belong to only one instance of the relevant associations:

$$\forall\, ec \in AssociationEndExclusionConstraint \bullet$$
$$\forall\, ae1,\ ae2 \in assocExclConst[\{ec\}] \bullet$$
$$\text{dom}\ (linkAssociation(assocAssocEnd1(ae1)^{-1}$$
$$\cup\ assocAssocEnd2(ae1)^{-1}))$$
$$\cap\ \text{dom}\ (linkAssociation(assocAssocEnd1(ae2)^{-1}$$
$$\cup\ assocAssocEnd2(ae2)^{-1}\ )) = \varnothing .$$

7 Discussion

This paper presents metamodels, as class diagrams, for the main concepts required to model the static aspects of information systems in the UML. For reasons of space, it does not present all the invariants used to express the metamodels in B. Similarly, it does not present the specifications as a B machine. This would list, for example, all given sets in a *SETS* clause, and the conjoined invariants in an *INVARIANT* clause.

A B machine specification expresses both diagrammatic structure and the additional semantics of the metamodel. This is seen as a significant advantage over OMG's diagrams and OCL constraints. The OCL must be read in the context of the diagrams, and, although OCL support tools are emerging (see http://www.klasse.nl/ocl/index.htm), the mutual consistency of diagrams and OCL cannot be checked.

The diagrammatic and OCL semantics cannot define the creation, modification and deletion of model elements, whereas the B machine could be extended with operations to create, modify and delete semantically-correct components of the IS class metamodel. Such a machine would add semantic rules such as the requirement that a class cannot exist in isolation (ie from a hierarchy or association; the initial creation would have to be of two associated classes). The operations could be demonstrated (by proof using one of the two B Method tools[3]) to respect the invariant. A semantically-correct IS UML tool could then be formally generated from the B specification.

The paper only considers the semantics of UML class models for IS. The authors have also explored the behavioural model semantics, focusing on a subset of UML notations for state diagrams and collaboration diagrams.

[3] Atelier-B, see http://www.atelierb.societe.com/ and The B Tool, see B-Core(UK) Ltd, http://www.b-core.com/

References

1. J-R. Abrial. *The B-Book*. CUP, 1996.
2. S. S. Alhir. Extending the Unified Modeling Language (UML). Available on line, home.earthlink.net/~ salhir/index.html, January 1999.
3. J. Alvarez, A. Evans, and P. Sammut. MML and the metamodel architecture. In *Workshop on Transformations in UML, Fourth European Joint Conference on Theory and Practice of Software (ETAPS'01), Genova, Italy*, April 2001.
4. C. Atkinson and T Kühne. Strict profiles: Why and how. In *UML 2000 - The Unified Modeling Language. Advancing the Standard, York, UK*, volume 1939 of *LNCS*. Springer Verlag, October 2000.
5. J-M. Bruel and R.B.France. Transforming UML models to formal specifications. In *UML 1998 - The Unified Modeling Language*, volume 1618 of *LNCS*. Springer Verlag, 1998.
6. A. Clark, A. Evans, S. Kent, S. Brodsky, and S. Cook. A feasibility study in rearchitecting UML as a family of languages using a precise OO meta-modeling approach. Technical report, pUML Group and IBM, September 2000. Available on line, www.cs.york.ac.uk/puml/papers/.
7. A. Clark, A. Evans, S. Kent, and P. Sammut. The MMF approach to engineering object-oriented design languages. In *LTDA2001: Workshop on Language Descriptions, Tools and Applications*, April 2001.
8. T. Clark and A. Evans. Foundations of the unified modeling language. In *Second Northern Formal Methods Workshop, Ilkley, UK*, volume 1241 of *LNCS*. Springer Verlag, 1997.
9. S. Cook. The UML family: Profiles, prefaces and packages. In *UML 2000 - The Unified Modeling Language. Advancing the Standard, York, UK*, volume 1939 of *LNCS*. Springer Verlag, October 2000.
10. S. Cook, A. Kleppe, R. Mitchell, B. Rumpe, J. Warmer, and A. Wills. Defining UML family members using prefaces. In C. Mingins and B. Meyer, editors, *Technology of Object-Oriented Languages and Systems, TOOLS'99 Pacific*. IEEE Computer Society, 1999.
11. Rational Software Corporation. *Unified Modeling Language version 1.1*. Rational Inc., July 1997. available on line, www.rational.com/uml.
12. D. D'Souza, A. Sane, and A. Birchenough. First-class extensibility for UML - packaging of profiles, stereotypes and patterns. In *UML 2000 - The Unified Modeling Language. Beyond the Standard*, volume 1723 of *LNCS*. Springer Verlag, 1999.
13. P. Facon, R. Laleau, A. Mammar, and F. Polack. Formal specification of the UML metamodel for building rigorous CAISE tools. Technical report, Laboratoire CEDRIC, Paris, September 1999.
14. P. Facon, R. Laleau, and H. P. Nguyen. From OMT diagrams to B specifications. In M. Frappier and H. Habrias, editors, *Software Specification Methods. An Overview Using a Case Study*, pages 57–77. Springer, 2000.
15. P. Facon, R. Laleau, H. P. Nguyen, and A. Mammar. Combining UML with the B formal method for the specification of database applications. Technical report, Laboratoire CEDRIC, Paris, September 1999.
16. Object Management Group. *OMG Unified Modeling Language Specification(draft)*. OMG, April 1999. BetaR1 Release, available on line, www.rational.com/uml.
17. R. Laleau and N. Hadj-Rabia. Génération automatique de spécifications formelles VDM á partir d'un schéma conceptuel de données. In *INFORSIDS'95, Grenoble, France*, June 1995.

18. R. Laleau and A. Mammar. A generic process to refine a B specification into a relational database implementation. In *ZB2000: Formal Specification and Development in Z and B*, volume 1878 of *LNCS*, pages 22–41. Springer Verlag, August–September 2000.
19. R. Laleau and A. Mammar. An overview of a method and its support tool for generating B specifications from UML notations. In *ASE: 15th IEEE Conference on Automated Software Engineering, Grenoble, France*. IEEE Computer Society Press, September 2000.
20. F. Monge. Formalisation du méta modèle des méthodes graphiques d'analyse et conception orientées objet. Master's thesis, Institut d'Informatique d'Entreprise, Conservatoire National des Arts et Métiers, Evry, September 1997.
21. H. P. Nguyen. *Dérivation de spécifications formelles B à partir de spécifications semi–formelles*. PhD thesis, Laboratoire CEDRIC, Conservatoire National des Arts et Métiers, Evry, December 1998. available from www.iie.cnam.fr/~ laleau/thesePHNguyen.ps-gz.
22. S. H. Nienhuys-Cheng. Classification and syntax of constraints in binary semantical networks. *Information Systems*, 15(5), 1990.
23. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Guide*. Addison-Wesley, 1998.
24. M. Sahbani. Formalisation des relations de composition et d'agrégation des modèles sémantiques. Master's thesis, Institut d'Informatique d'Entreprise, Conservatoire National des Arts et Métiers, Evry, September 1999.
25. L. T. Semmens, R. B. France, and T. W. G. Docker. Integrated structured analysis and formal specification techniques. *The Computer Journal*, 35(6), 1992.

Appendix

In UML1.3, the semantic models separate structure and behaviour (figure 5, drawing together semantics from UML1.3 figures 2.5 and 2.8 [16]). For example, the subclasses of *ModelElement* include *Features* and *GeneralizableElement*. Classes are a second-level specialisation of *GeneralizableElement*, whilst attributes and operations are second-level specialisations of *Features*. The UML1.3 metamodels have only indirect associations between *attribute* and *association* (figure 6, based on UML1.3 figures 2.6 and 2.8 [16]). This makes it difficult to express the semantics of structural features such as association qualifiers, which must be a set of attributes from the *Class* of the relevant *AssociationEnd*. Note that in UML1.1 [11], and in the UML Reference Manual [23, p161], association qualifiers are represented as an attribute of *AssociationEnd*. This does not permit expression of the semantics of qualification, via sets of attributes.

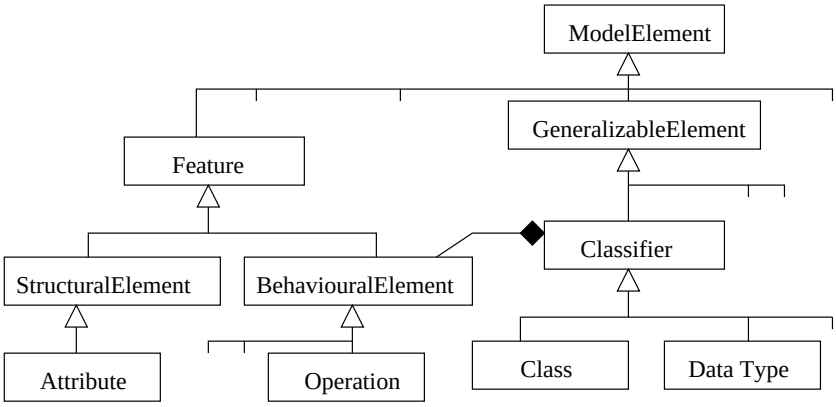

Fig. 5. Extracts of the OMG Metamodels showing the Use of Subclasses to Distinguish Semantic Element Groups

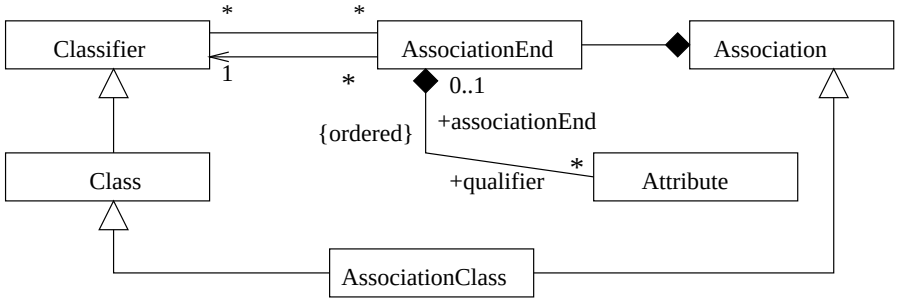

Fig. 6. Extracts of the OMG Metamodels showing the Semantics of Associations and Qualifiers