Checking the Consistency of UML Class Diagrams Using Larch Prover

Pascal André[1], Annya Romanczuk[2], Jean-Claude Royer[1], and Aline Vasconcelos[2]

[1] Équipe Génie Logiciel, Méthodes et Spécifications Formelles
IRIN - Université de Nantes
2, rue de la Houssinière - BP 92208
44322 Nantes Cedex 03
Pascal.Andre,Jean-Claude.Royer@irin.univ-nantes.fr
[2] Groupe Systèmes et Langages à Objet
École des Mines de Nantes
4, rue Alfred Kastler - BP 20722
F-44307 Nantes Cedex 3
Annya.Romanczuk@emn.fr,alinepires@censa.com.br

Abstract. This article introduces a formal semantics for UML based on algebraic abstract data types. We currently consider only class and object diagrams. These diagrams include class structures, associations, multiplicities, constraints, instances as well as specialization relationships. We give a formal semantics for each of these elements by interpreting the structure of a class as an abstract data type, associations as values of type `Association`, and specialization as structural projection. We show that a tool like Larch Prover is able to support proofs over UML diagrams. We use the critical pair computation to find out inconsistencies. Several different inconsistencies of class diagrams are shown on a library example.

Keywords. Abstract Data Type, Algebraic Specification, Class Diagram, Consistency, Critical Pair Computation, Larch Prover, UML.

1 Introduction

The Unified Modeling Language (UML) [Con99] has been designed to be a full standard notation for Object-Oriented Modelling. UML is a rather complete set of notations, including 9 diagrams that cover all the semantics of the object paradigm. The price to pay is an informal semantics: ambiguous concepts (*e.g.* the semantics of aggregation [BHS99]), concepts overlapping the diagrams (with sometimes a quite different semantics (*e.g.* class *vs* actor). As a consequence, the modelling is seldom complete and often includes many inconsistencies inside the diagrams or between them. The definition of a formal semantics for UML is a mandatory step for the normalization of UML to be mature, as underlined by the pUML group[1].

[1] http://www.cs.york.ac.uk/puml/home.html

Such a definition is a difficult task because the notation is rich, complex and informal. The class diagram being the reference point of the notation, any formalization must start with this diagram. The current proposals are mainly based on state-based languages like Z [FBLPS97,LB98]. We choose an algebraic approach because it is more abstract than state-based style languages. Another reason is the number of existing tools and theoretical frameworks. From an historical viewpoint, object-oriented languages and abstract data types (ADT for short) share a common ancestor: Simula [KD66]. The two techniques were developed independently, however some works have tried to merge them: FOOPS [GM87], and Eiffel [Mey88]. The main and first approach of a semantics for UML with ADT was [HHK98b]. It reuses a previous work about OMT [BC95]. The authors use the Larch Shared Language. The present work is related to these approaches but we focus on a more operational approach and the checking of consistency within the Larch Prover environment. There are also some other differences, mainly in the semantics given for association, composition and inheritance. We introduce a general semantic framework for UML class and object diagrams. The current work includes class related elements (class, association, constraints) and their specialization as well as objects and links. We assume the reader to be familiar with the corresponding UML notations.

A great part of this paper is devoted to the use of the Larch Prover [GG89] (LP) to check the correctness of class diagrams. The consistency of such diagrams is not easily checkable. One reason is the possible use of code annotations to denote constraints on the elements of the diagram. Code annotations may be written in programming languages or rather in OCL (Object Constraint Language). OCL is a kind of first order functional language with collections and iterator facilities. In this paper, code annotations will be LP expressions in order to lighten the translation process. The use of OCL with a similar translation to [HHK98c] would be a good way. The problem of consistency of such class diagrams is related to the consistency of first order predicate logic. This kind of problem is not decidable [Apt90]. It exists several useful approaches to detect either consistency or inconsistency. For example, techniques based on Herbrand's Theorem may detect inconsistency [DJ90], but they require process termination since this problem is semi-decidable. Here we use LP and under some hypotheses it can prove a system consistent for equational logic. These hypotheses concern the use of quantifier free formulas, the termination of the rewriting system and the termination of the Knuth-Bendix completion algorithm. In practice it is difficult to prove consistency, and it is often easier to detect inconsistencies with the critical pair computation.

The structure of the paper is the following. Section 2 describes the context: algebraic specifications of data types and Larch Prover. We briefly explain the method we use is based on the critical pair computations. The third Section gives the main elements of our translational semantics of UML class diagram into algebraic specifications. In Section 4 we illustrate our method on different examples where we have detected inconsistencies. We conclude by lessons about these experiments, some related works and future extensions.

2 Algebraic Specifications and Larch Prover

Algebraic specifications have been used for the definition of ADT and the formal specification of safety systems for a long time. The reader will find a rather comprehensive introduction and survey about these formal specification techniques in [Wir90].

Larch Prover [GG89,GH93] (LP for short) is a theorem prover based on the Larch Shared Language. It allows one to define algebraic specifications, to use rewrite rules and to prove properties. However it support neither partial algebras nor genericity. Some modifications are done to our specifications in order to fix these problems. The genericity problem is simply fixed by a copy and replace mechanism. Partial operations are left unspecified, we generate axioms with terms which are assumed to be well-defined. LP has a set of proof tools, mainly they are: rewriting, critical pair computation, Knuth-Bendix completion procedure, proof by induction, proof by contradiction, and proof by case. LP supports first order predicate calculus with equality.

An algebraic specification of a data type is composed of three parts:

- a heading, containing informations about the module: the name (or sort) of the type, the imported modules, and the constructor names (or generators),
- a signature which describes the operation syntax, and
- the axioms which describe the operation semantics.

An example, with the `Association` ADT, is depicted in Figure 2. Note that operations of a defined type are classified into internal operations and observers. An internal operation is an operation whose resulting type is the defined type. A generator is a particular internal operation needed to build the values of the type. We use an informal syntax close to the Larch Prover one. Axioms are conditional: `cond => t = u` where `t, u` are algebraic terms and `cond` is a conjunction of `ti = ui` equations. Larch Prover allows one to write first-order expressions where ˜is logical not, $\wedge$ is and, $\vee$ is or, `=>` is implication, `=` is equality. Existential (respectively universal) quantified propositions are possible with the `\E` (respectively the `\A`) prefix.

The general way, in LP, to check consistency is to compute the critical pairs. A *critical pair* is a potential ambiguity in a set of rules. It can be either a new fact forgotten in the system or an irremediable inconsistent fact (for instance `true = false`). The command `critical-pair <setr1> with <setr2>` computes critical pairs between two set of rewriting rules. The `complete` command uses the well-known Knuth-Bendix completion algorithm[2]. This algorithm computes all the critical pairs and adds them into the system. The process may stop with an inconsistency. When the process terminates without inconsistency, the system is consistent. Otherwise the process does not terminate, and we cannot say anything about the consistency. If the model does not contain any user code annotation, then our translation generates an algebraic system which is terminating and consistent. Carefully note, consistency is related to equational logic

[2] More precisely it is the Peterson-Stickel extension of this algorithm.

not to predicate calculus. Strictly speaking, it needs a logical system with only quantifier free equations and rewrite rules. However, first-order formulas can be transformed (by skolemization) into quantifier free formulas.

3 Translation of UML Class Diagrams

In this section we sketch a translational semantics for UML class diagrams. More details and additional elements (aggregation, composition, association as class, abstract classes, instance management or the specification file organization) can be found in [Roy99,ARRV00]. A class diagram is translated into algebraic definitions. We do not take into account code inheritance and side effects, hence a purely functional approach with identities is pertinent here. The existing instances of a class equate the values of the ADT associated to that class. To define the semantics of a class, two types are needed: the effective type and the maximal type. The *effective type* is the set of the effective instances of the class. The *maximal type* is the set of all instances of a class coping with subclasses. This provides an approach where implicit and explicit descriptions of instances and associations are possible. In order to define the mapping rules from class diagrams to algebraic ADTs, the syntax and semantics of each diagram element is evaluated. We give in the following sections the description of some of these mapping rules and the corresponding resulting algebraic constructions.

3.1 From Class to ADT

We assume that classes are formalized by ADTs. Basic types are supposed to exist (imported ADTs). To explicit the connection between ADT and Object-Oriented programming, we reuse the formal class principles [ACR94]. A UML object has an implicit identity. The identity value of an instance of class `C` is defined by the `IdC` associated type. An implicit hypothesis is: two values represent the same object if and only if they have the same identity. As in programming language there are two equalities:

- an *object equality* (`eq`) which is based on identity equality, and
- a *semantic equality* (`equal`) which is based on equality of attribute values.

As usual, the first one implies the second one. Taking class `C`, the following mapping rules are defined. A single `newC` constructor represents the instantiation process. Instance attributes are translated into argument types in the constructor signature. For each argument type of the constructor, primitive observers, called *field selectors*, are defined.

```
declare operator  newC      : IdC, T1, ..., Tn -> C
declare operator  identity  : C                -> IdC
declare operator  sel1      : C                -> T1
declare operator   ...
declare operator  seln      : C                -> Tn
```

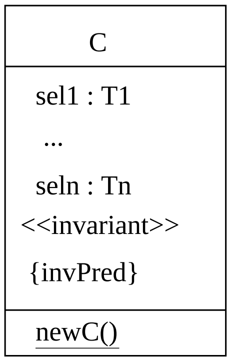

Fig. 1. A Simple UML Class Description

Where `sel1:T1, ..., seln:Tn` are the typed attributes of class `C`. By definition, primitive observers are operations related to the main aspect of the ADT which semantics is defined in terms of the constructor.

```
assert  identity(newC(id, X1, ..., Xn)) = id;
assert  sel1(newC(id, X1, ..., Xn)) = X1;
assert   ...
assert  seln(newC(id, X1, ..., Xn)) = Xn;
```

The `invPred` is translated as a condition of the `newC` constructor. Thus for each occurrence of the `newC` constructor in each axiom we must generate an instantiation of the `invPred` and put it in the condition part. Object equality is based on identity equality and semantic equality is recursive structural equality (but for identity).

```
declare operator  __\eq__ :  C, C -> Boolean
assert  aC1 \eq aC2 = equal(identity(aC1), identity(aC2));
declare operator  equal   :  C, C -> Boolean
assert  equal(newC(id, X1, ..., Xn), newC(jd, Y1, ..., Yn))
                            = equal(X1, Y1) /\ ... /\ equal(Xn, Yn)
```

3.2 Association and Multiplicity

The following ADT describes a part of a general type which potentially defines any association between generic types `A` and `B`.

The `\eq` operator defines an intensional equality between associations. Of course this type can be completed with other operators, for instance an `equal` operator. An association is a value of type `Assoc[A, B]` with a name, a set of links and association properties (multiplicity, attributes, etc). We have a higher-order association type which allows one to define many general associations, computing multiplicity, searching links or searching left or right associated objects. It may be specialized and adapted to different situations by adding operations

```
Specification: Assoc[A, B]
Generic: A, B
Sort: Assoc[A, B]
Use: Nat, Boolean, A, B, String
Constructors: void, addLink
Variables: s : String, b, b1 : B, a, a1 : A,  Xab, Yab  : Assoc[A,B]

Signature:
  void :             String                   -> Assoc[A, B]
  name :             Assoc[A, B]              -> String
  addLink :          Assoc[A, B], A, B        -> Assoc[A, B]
  isEmpty :          Assoc[A, B]              -> Boolean
  isLinked :         Assoc[A, B], A, B        -> Boolean
  isRightLinked :    Assoc[A, B], A           -> Boolean
  rightCardinality : Assoc[A, B], A           -> Nat
  __ \eq __      :   Assoc[A, B], Assoc[A, B] -> Boolean

Axioms:
 name(void(s)) = s;
 name(addLink(Xab, a, b)) = name(Xab);

 Xab \eq Yab = equal(name(Xab), name(Yab));

 isEmpty(void(s));
 ~(isEmpty(addLink(Xab, a, b)));

~(isLinked(void(s), a, b));
isLinked(addLink(Xab, a, b), a1, b1) = (((a \eq a1) /\ (b \eq b1))
                                       \/ isLinked(Xab, a1, b1));

~(isRightLinked(void(s), a));
isRightLinked(addLink(Xab, a, b), a1) = ((a \eq a1)
                                        \/ isRightLinked(Xab, a1));

rightCardinality(void(s), a) = 0;
~(a \eq a1) => rightCardinality(addLink(Xab, a1, b1), a)
                                         = rightCardinality(Xab, a);
(a \eq a1) /\ isLinked(Xab, a, b1) =>
 rightCardinality(addLink(Xab, a1, b1), a) = rightCardinality(Xab, a);
(a \eq a1) /\ ~isLinked(Xab, a, b1) =>
rightCardinality(addLink(Xab, a1, b1), a) = 1+rightCardinality(Xab, a);

End.
```

Fig. 2. The Assocition ADT

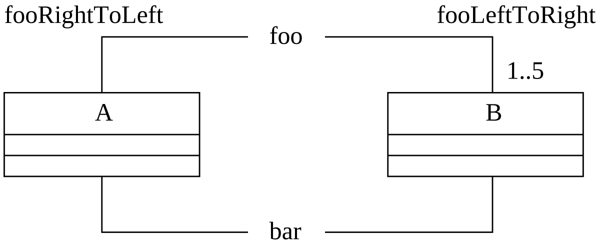

Fig. 3. Examples of Association

and axioms. Of course it is possible to extend this type to n-ary associations. To define a particular association we declare a new value with a name.

```
declare operator fooAssoc : -> Assoc[A, B]
assert  name(fooAssoc) = "foo";
```

One example named `bar` with explicit links is:

```
declare operator barAssoc : -> Assoc[A, B]
assert
 barAssoc = addLink(oneA, oneB, addLink(oneA, otherB, void( "bar")));
```

To assert multiplicity is simple using numeric relations, for instance:

```
assert    inf(rightCardinality(fooAssoc, aA), 5)
       /\ inf(1, rightCardinality(fooAssoc, aA));
```

where `inf` stands for the “lesser or equal” relation.

3.3 Navigation and Role

In order to express explicit navigation, it is necessary to define a new operator. The same schema allows the definition of roles for the association.

```
declare operator  fooLeftToRight : A -> Set[B]
assert  fooLeftToRight(aA) = rightObjects(fooAssoc, aA)
```

3.4 Code Annotation and Constraint

UML guide suggests to use OCL or a programming language to express code annotations. Our code annotations are LP expressions because they may easily be integrated into the target LP specification. It is possible to define general constraints on classes, operations or relations using code annotations. As an example we define a general `rightXor` constraint on two associations of type `A-B` and `A-C` respectively.

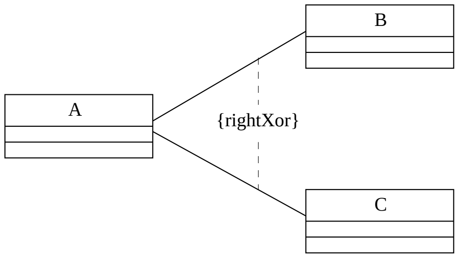

Fig. 4. A rightXor Constraint

```
declare variable ABass : Assoc[A, B], CDass : Assoc[A, C], aA : A
declare operator rightXor : Assoc[A, B], Assoc[A, C] -> Boolean
assert    ~(ABass \eq ACass) => rightXor(ABass, ACass) =
           ((~isRightLinked(ABass, aA) /\ isRightLinked(ACass, aA))
            \/ (isRightLinked(ABass, aA) /\ ~isRightLinked(ACass, aA)))
```

Note that this definition assumes the two associations are distinct. A common mistake is to forget such conditions. If `B=C`, the two associations may be the same, hence the `rightXor` definition would not be consistent. In the previous example, axiom writing is only limited by first order expressions. Since explicit quantifiers may be used (see examples later) we achieve a great degree of expressiveness.

3.5 Inheritance

Specialization is made up of two parts: structural inheritance (attributes and associations) and behavioural inheritance (operations). In this paper we only deal with the structural part. Structural specialization includes: adding new attributes or associations, strengthening the invariant constraint or the multiplicity intervals, redefining the attributes or associations. However, we assume that the multiplicity intervals are not redefined, but each interval can be reduced. We consider an extended UML approach with multi-covariant redefinitions. Therefore type problems may arise in this framework, even in a purely functional approach [Cas97]. These problems are well-known and some solutions exist [BCC+96,Roy98]. Once translated into our algebraic framework, we assume that our specifications do not carry type-checking problems.

Subclass Translation. Let us consider the specialization between class `C` and `SubC` one of its subclasses. The `SubC` defines a proper structure: `sub1: S1, ..., subm : Sm`. We translate the class to get the following ADT:

```
Specification: SubC
Sort: SubC
Use: C, IdSubC, T1, ..., Tn, S1, ..., Sm
Constructors: newSubC
Variables: id : IdSubC, X1 : T1, ..., Xn : Tn, Y1 : S1, ..., Ym : Sm
```

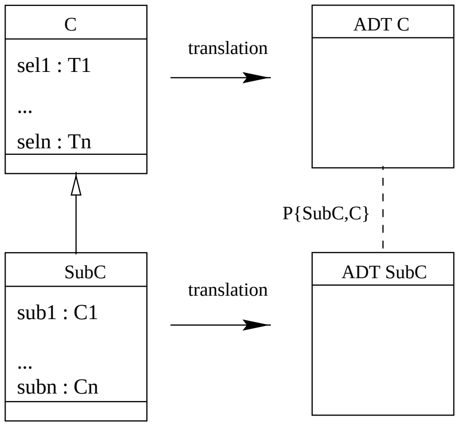

Fig. 5. Structural Specialization

```
Signature:
 newSubC :  IdSubC, T1, ..., Tn, S1, ..., Sm -> SubC
 identity : SubC -> IdSubC
 sel1 :     SubC -> T1
 ...
 seln :     SubC -> Tn
 sub1 :     SubC -> S1
 ...
 subm :     SubC -> Sm
 __ \eq __ : SubC, SubC -> Boolean
 equal :     SubC, SubC -> Boolean
Axioms:
 identity(newSubC(id, X1, ..., Xn, Y1, ..., Ym)) = id;
 sel1(newSubC(id, X1, ..., Xn, Y1, ..., Ym)) = X1;
 ...
 seln(newSubC(id, X1, ..., Xn, Y1, ..., Ym)) = Xn;
 sub1(newSubC(id, X1, ..., Xn, Y1, ..., Ym)) = Y1;
 ...
 subm(newSubC(id, X1, ..., Xn, Y1, ..., Ym)) = Ym;
End.
```

As previously explained in translation of class `C`, we also adds `eq` and `equal` axioms to the ADT associated to class `SubC`. In fact, the specialization of equalities is subtle because object equalities are symmetric and polymorphic. Some solutions exist in [Ing86,BCC+96]. Here we simply add declarations to take into account parameter polymorphism.

```
declare variable Xc : C, Xs : SubC
declare operator __ \eq __, equal : C, SubC -> Boolean
declare operator __ \eq __, equal : SubC, C -> Boolean
assert    ~(Xc \eq Xs);
```

```
assert    ~(Xs \eq Xc);
assert    equal(Xc, Xs) = equal(sel1(Xc), sel1(Xs)) /\ ...
                              /\ equal(seln(Xc), seln(Xs));
assert    equal(Xs, Xc) = equal(sel1(Xs), sel1(Xc)) /\ ...
                              /\ equal(seln(Xs), seln(Xc));
```

Structural Projection. An instance of `SubC` can be viewed as an instance of `C`. We formalize this relation by a *structural projection* which comes from a previous work [Roy92]. The interpretation of specialization is done according the Figure 5. Specialization of structure defines a correspondence between the arguments of the ADT constructors. We assume this correspondence is: the first n arguments (except identity) of the `newSubC` constructor corresponds to the n arguments (except identity) of the `newC` constructor. When a subclass is defined we add the definition of the structural relation between the ADTs corresponding to the classes. If `SubC` inherits from `C` then we define the `P{SubC,C}` relation between the ADTs associated to `SubC` and `C`:

```
P{SubC,C}(newSubC(id, X1, ..., Xn, Y1, ..., Ym),
      newC(jd, Z1, ..., Zn)) = equal(X1, Z1) ∧ ... ∧ equal(Xn, Zn).
```

This defines a partial ordering on abstract data types parallel to the class specialization relationship. This definition is automatically computed once the specialization relations are known. This relation verifies:

$$\forall s_1, s_2 : SubC,\quad (equal_C(s_1, s_2) \iff \exists c : C, P\{SubC, C\}(s_1, c) \ \wedge \ P\{SubC, C\}(s_2, c))$$

where $\mathtt{equal}_C$ is the equality predicate for two instances of class `C`. It means that the specialization relation defines an equivalence relation on the subclass instances. Furthermore this relation yields the following property:

$$\forall s : SubC,\ \forall c_1, c_2 : C, P\{SubC, C\}(s, c_1) \wedge P\{SubC, C\}(s, c_2) \Longrightarrow equal_C(c_1, c_2)$$

Uniqueness of the superclass instances is based on semantic equality, namely the `equal` predicate. These properties express an isomorphism between the equivalence classes associated to the $\mathtt{equal}_C$ relation in class `C` and subclass `SubC`. We consider `P{SubC,C}` as a relation between class `C` and `SubC` to be more uniform with associations. Note that it can be seen as a total function from `SubC` to `C`. In general it is a many-to-one function and it is not onto.

Association and Specialization. Whenever an association is defined, we expect it to be inherited in some way. This has only effect on structural specialization. An adequate notion of association specialization seems also required. A more comprehensive discussion about this topic exists in [Roy99].

In Section 3.2 we did not handle class polymorphism, but only *monomorphic associations* between types. From now on, such an association will be noted $assoc^m$. A general definition of a *polymorphic association* ($assoc^p$) between two classes is a set of monomorphic associations:

$$assoc^p[A, B] \ = \biguplus_{A^i, B^j \ \leq \ A,B} assoc^m[A^i, B^j]$$

where A^i and B^j are subclasses of (respectively) A and B, and $assoc^m[A^i, B^j]$ is a monomorphic association between the types associated to A^i and B^j. We give a semantics of inclusion to the specialization of associations. Let $asub$ and $assoc$ be two associations with the same name, and $A', B' \leq A, B$, an association $asub^p[A', B']$ is a specialization of another $assoc^p[A, B]$ iff $asub^p[A', B'] \subseteq^p assoc^p[A, B]$. Inclusion of polymorphic associations is defined as:

$$asub^p[A', B'] \subseteq^p assoc^p[A, B]$$

$$\Longleftrightarrow$$

$$\forall A'^i, B'^j \leq A', B', asub^m[A'^i, B'^j] \subseteq^m assoc^m[A'^i, B'^j]$$

where $\subseteq^m$ stands for inclusion association. Much work still has to be done in order to get more useful and specific definitions, and also to relate them to association classes.

3.6 Tool Integration

To allow the automatic generation of algebraic specifications from an UML diagram, we developed an UML case tool carrying the mapping rules integrated in its context. The case tool was built [dV99] in Graphtalk meta-tool[3]. We have not yet translated the OCL language into algebraic specifications, but [HHK98c] demonstrates its feasibility.

4 Checking UML Diagrams Consistency Using LP

As we explained before, we use LP to detect inconsistencies in class diagrams. We often need to prove auxiliary facts and LP is of great help in such cases. In this Section, we illustrate the use of LP to check class diagrams. Of course in some examples the problem is quite obvious because the wrong elements have been extracted from larger specifications. Ordinary diagrams contain many classes within different packages so that a tool is necessary to detect the problems.

4.1 Examples about Association Consistency

The following examples present inconsistencies with associations, constraints and multiplicities. There are different kinds of consistency problems in UML diagrams, like problems with types of associations (*e.g.* an inclusion constraint needs that association types are the same). Some problems can be detected by type-checking or multiplicity-checking. However, this is not generally sufficient because the specifier is able to define any kind of constraint, and some constraints need a more complex checking. This is illustrated by the last example of this Subsection.

[3] Graphtalk meta-tool is a product of Computer Sciences Corporation.

A Constraint and Multiplicity Problem. The practitioner usually writes separately the multiplicities and the constraints. Sometimes, their conjunction is inconsistent. Let consider the example of Figure 6, where `A = Person`, `B = Book`, `assoc1 = author` and `assoc2 = reviewer`. All the people are both author and reviewer of a book, but one cannot review his proper book. Using the previous association type we may define the `assoc1=author` and `assoc2=reviewer` associations. It can be simply expressed by the `P1` existential proposition:

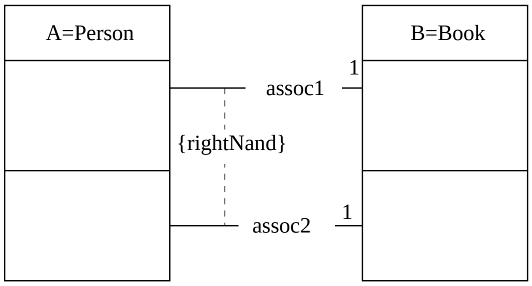

Fig. 6. A Simple Example

```
declare variable Xab, Yab : Assoc[A, B], a : A
assert \E Xab \E Yab (~(Xab \eq Yab) /\ (rightCardinality(Xab, a) = 1)
              /\ (rightCardinality(Yab, a) = 1) /\ rightNand(Xab, Yab))
```

The `Nand` constraint may then be defined by proposition `P2`:

```
declare operator rightNand : Assoc[A,B], Assoc[A,B] -> Bool
assert ~(Xab \eq Yab) => rightNand(Xab, Yab) =
                 ~(isRightLinked(Xab, a) /\ isRightLinked(Yab, a))
```

A Simple Proof. Below is a simple proof done with the help of the Larch Prover. The assertion says that if the multiplicity is equal to 1 then there is a link in the association.

```
prove equal(rightCardinality(Xab, a), 1) => isRightLinked(Xab, a)
% these are LP comments
% The above command tries to resume a proof by induction
  resume by induction on Xab
     <> basis subgoal
% This means ok for basis case
    [] basis subgoal
    <> induction subgoal
% Try to prove the induction step by implication
    resume by =>
      <> => subgoal
% Try to prove by case
```

```
      res by case  identity(a1c) \eq identity(ac)
        <> case identity(a1c) \eq identity(ac)
        [] case identity(a1c) \eq identity(ac)
% Try to prove by the opposite case
        <> case ~(identity(a1c) \eq identity(ac))
% Addition of a trivial lemma
        assert ~(identity(ac) \eq identity(a1c))
% Critical pair computation to finish the proof
        crit as* with as*
        [] case ~(identity(a1c) \eq identity(ac))
      [] => subgoal
    [] induction subgoal
  [] conjecture
qed
```

Proofs are usually not automatic, an expert user must choose the way to process them. We can now show that the previous UML model is not consistent by defining two associations `as1(a), as2(a)`:

```
declare operator as1, as2 : A -> AssocAB
% eliminations of \E
fix Yab as as2(a), Xab as as1(a) in P1
% elimination of \A
instantiate Xab by as1(a), Yab by as2(a) in P2
LP says that the system becomes inconsistent
```

It illustrates one fact: the `Nand` constraint is not compatible with two multiplicities of 1. The previous example easily extends to a class A `rightNand` associated to class B1, B2, .. Bn. There are similar problems with `xor` or other constraints. This gives rules to define a kind of multiplicity-checking. Note that there is a more straightforward way which makes explicit use of the critical pair computation as in the following examples.

Inconsistency without Multiplicity Problem. We illustrate a simple example with an inconsistency, but multiplicity-checking would not find it. In this

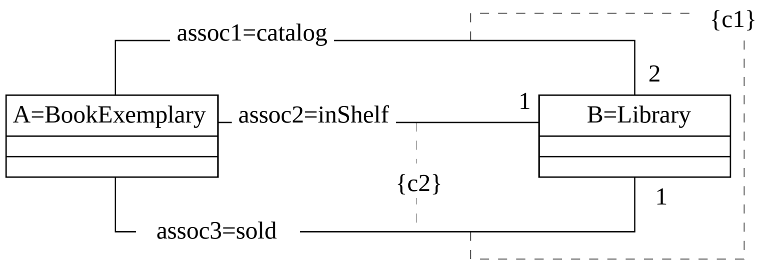

Fig. 7. Inconsistency Without Multiplicity Problem

example C1 is a constraint which states that each link in assoc1 appears either in assoc2 or in assoc3 and not in both associations. The constraint C2 states that the associations assoc2 and assoc3 are equal. It may seem that this is a consistent example, but it is not. Since multiplicities are strictly greater than zero, the associations are not empty. But the C1 relation and the equality between assoc2 and assoc3 imply that these associations are empty. Below are the corresponding axioms, auxiliary facts and the inconsistency proof.

```
declare operator inclus : AssocAB, AssocAB -> Bool
assert inclus(void(s), Yab);
assert inclus(addLink(Xab, a, b), Yab) = (isLinked(Yab, a, b)
                                         /\ inclus(Xab, Yab))
declare operator emptyInter : AssocAB, AssocAB -> Bool
assert emptyInter(void(s), Yab);
assert emptyInter(addLink(Xab, a, b), Yab) = (~isLinked(Yab, a, b)
                                            /\ emptyInter(Xab, Yab));
declare operator C1 : AssocAB, AssocAB, AssocAB -> Bool
assert  C1(Xab, Yab, Zab) = (inclus(Yab, Xab) /\ inclus(Zab, Xab)
                                             /\ emptyInter(Yab, Zab));
declare operator C2 : AssocAB, AssocAB -> Bool
assert C2(Yab, Zab) = (inclus(Yab, Zab) /\ inclus(Zab, Yab))
declare operator assoc1 : -> AssocAB
declare operator assoc2 : -> AssocAB
declare operator assoc3 : -> AssocAB
assert rightCardinality(assoc1, a) = 2
assert rightCardinality(assoc2, a) = 1
assert rightCardinality(assoc3, a) = 1
assert C1(assoc1,assoc2,assoc3)
assert C2(assoc2, assoc3)
assert ~(equal(rightCardinality(Xab, a), 0) => ~isEmpty(Xab)
assert inclus(Yab, Zab) /\ emptyInter(Yab, Zab) => isEmpty(Yab)
complete
% ...
Formula exemple.19, false, is inconsistent.
```

4.2 Examples with Specialization

Specialization adds also some potential problems with multiplicity and specialized association.

Wrong Multiplicity or Wrong Constraint. The specialization semantics of Section 3.5 implies that the multiplicity intervals of the specialized association must be included or equal to the ones of the base association. Hence Figure 8, example a is inconsistent. Furthermore, some constraints are inconsistent with specialization, for instance Figure 8, example b. We may check these two examples and detect inconsistencies. In case 8.a a critical pair computation finds the problem, but a complete command will diverge. In case 8.b a complete

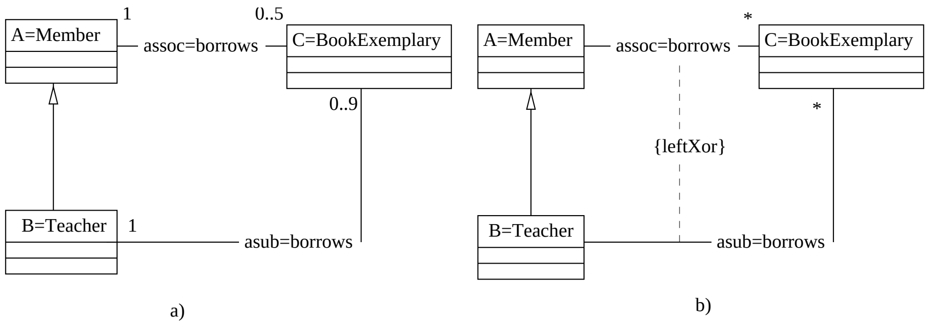

Fig. 8. Wrong Examples with Specialization

command will prove that `asub` has to be empty, and if not an inconsistency is detected.

```
set name ex
%assoc = assoc1+assoc2
declare operator assoc1 : -> AssocAB
declare operator assoc2 : -> AssocAsubB
% asub
declare operator asub : -> AssocAsubB
assert isLinked(asub, anasub, b) => isLinked(assoc2, anasub, b)
% the left xor constraint
assert ((~isLinked(assoc1, a, b) /\ ~isLinked(assoc2, anasub, b)
        /\ isLinked(asub, anasub, b)) \/ ((isLinked(assoc1, a, b)
       \/ isLinked(assoc2, anasub, b)) /\ ~isLinked(asub, anasub, b)))
complete
% The system is complete.
declare operator unasub : -> Asub
declare operator unb : -> B
assert isLinked(asub, unasub, unb)
Added 1 fact named ex.6 to the system.
Formula ex.6, false, is inconsistent.
```

Constraint and Specialized Associations. We present here a more complex example where an association (`assoc`) is specialized twice.

```
% assoc = assoc1+assoc2+assoc3
declare operator assoc1 : -> AssocAB
declare operator assoc2 : -> AssocCB
declare operator assoc2 : -> AssocDB
assert equal(rightCardinality(assoc1, a) + rightCardinality(assoc2, c)
                                         + rightCardinality(assoc2, d), 2)
% asub1
declare operator asub1 : -> AssocCB
assert isLinked(asub1, c, b) => isLinked(assoc2, c, b)
```

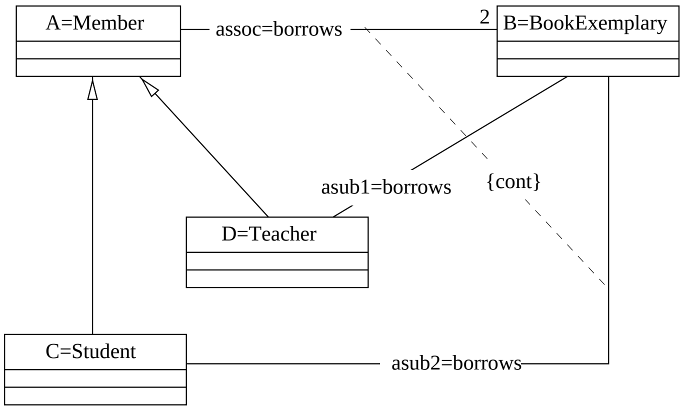

Fig. 9. Example with Constraint and Specialized Associations

```
% asub2
declare operator asub2 : -> AssocDB
assert isLinked(asub2, d, b) => isLinked(assoc3, d, b)
% fact for cont
assert equal(c, d) /\ isLinked(asub1, c, b) /\ isLinked(asub2, d, b)
 => (\E a equal(a, c) /\ isLinked(assoc1, a, b))
```

In this context, we detect an inconsistency if the redefined associations contains one association with two objects equal with respect to the `A` equality. However, this inconsistency is more difficult to prove.

4.3 Synthesis

Using our algebraic approach, we have checked several kinds of inconsistencies:

- multiplicity problem with derived association,
- inconsistency between multiplicity and constraint (see Figure 6),
- inconsistency without neither multiplicity nor type problem (see Figure 7),
- problem with multiple and disjoint specialization,
- problem with circular composition,
- problems with specialization, multiplicity and constraint (see Figure 8),
- problem with constraint and specialized association (see Figure 9).

We have studied different examples and proved their inconsistency, but the results are under the influence of the form of the axioms and the properties known by the system. In the previous examples, the following schema is able to detect any inconsistency:

1. translate the diagram into LP,
2. load the files,
3. prove some useful consequences and add them in the system, and

4. check the consistency with `complete` or `critical-pair` commands.

However, this is still far away from an automatic strategy to check inconsistency. On the other hand, our approach already gives us some advantages. One has a general approach (for expert only) which allows to define any constraint, to prove properties and to detect inconsistency. Our approach can be adapted to different contexts and may be used to improve or to give formal semantics to UML concepts. One can define and prove simple static checking based on the compatibility between multiplicity and constraint. For instance `Nand` is not compatible with 1-1, composition or transitivity imply that multiplicity are in relation of integer times, and the inclusion constraint implies that multiplicities are lesser or equal. Sometimes, an automatic strategy to detect problems exists, and it can be implemented in UML tools. Clearly, type-checking is a first and simple level of checking. It is able, for instance, to detect problems in derived associations or ill-written constraints. Another control is to check the multiplicity consistency with association constraints. [CLN99] described a language suitable to Entity-Relationships and DataBase models together with a decidable logic for it. This work can be useful to check some parts of the UML static diagrams (cardinality and predefined constraints). However, [CLN99] deals only with the non recursive structural aspects of objects and does not include any feature for the specification of behavioural properties of objects. This is insufficient because invariant, derived association, constraint are behavioural specifications of UML class diagrams. We suspect that a similar checking would be able to prove the consistency if there is only set constraints between association regardless of recursive data types.

5 Related Work

One important source of information, the first in this area, is [BC95]. It is related to OMT and the use of the Larch Shared Language (LSL). LSL is an abstract language for specification, which is different from the Larch Prover tool language. The authors study in depth classes, associations, cardinality, aggregation and many aspects of OMT. They also consider notions of state and errors. Following this work, Hamie, Howse and Kent apply it to the UML in [HHK98b]. Using this framework they also study navigation and OCL [HHK98a]. They use the LSL in a modular way and they give semantics to some static and dynamic parts of UML, including state changes and time.

Our definition is closer to algebraic specifications for data types and therefore operations are quite easy to define. [HHK98b] have an abstract approach but they do not guarantee the usual properties about algebraic specifications such as consistency and sufficient completeness. For us, it is not sufficient to write formal specifications, we want to be able to check them. We have a constructive, more operational and more precise approach, as indicated by the choice of generator, the notion of sensible signatures, and the two equality functions. It is difficult, and even impossible, without any additional information, to prove properties in the [HHK98b] approach.

We interpret neither OCL nor dynamic diagram. Our current approach seems not to be adequate for the semantics of a dynamic diagrams. We plan to use other works based on TAG [AR99]. In our context, equality is important because, for example, composition needs object identity and inheritance needs semantic equality. We have an explicit instantiation process; instances and associations can be implicitly or explicitly defined. Associations in [HHK98b] are interpreted as two functions that map an object of one type to the set of associated objects of another type. In our approach associations are first class values, hence `Assoc[A, B]` is a true and constructive type. This is not true in [BC95,HHK98b]. We think this is more flexible and natural for UML, and that makes the integration of other constructions easier. For example, there is neither generic constraint nor association class in [HHK98b]. In this work there are no semantics for composition either. We also have a proper semantics for specialization which improves the one of [BC95] (reused in [HHK98b]). This former work defines the `simulate` function which asserts that a subclass is a subtype of another class by requiring that associated sorts are in subsort relationship. The `simulate` function maps an object identifier of the subclass to an object identifier of the super class that behaves like it. The [BC95] and [HHK98b] works take into account only the maximal type, in addition we formalize the effective type. This provides an approach where implicit and explicit descriptions of instances and associations are possible. We also propose a definition for the structural projection based on semantic equality. It allows us to define associations and inherited code in an operational way. Other related works are [HCRT99,FC99].

6 Conclusion

Our approach gives a formal semantics to UML class diagrams using abstract data types. It is a successful approach on this sub part of UML. Our work is complementary with previous works and it improves them in several ways: it suggests first-class association, it allows association constraints, it formalizes structural inheritance, it gives a useful and rigorous definitions to specialized association. We also show that the use of a tool like Larch Prover helps in the proof of properties and also can detect some ill-formed designs. Further work will concern methodologies to prove properties. We want to focus on an automatic detection of inconsistencies, and to be able to show schema equivalence. The use of LP is interesting, but actually it surely cannot handle very big examples. We consider that a practical limit should be around 1000 rules in LP. Therefore we have to optimize our translation schema and define new automatic strategies to check examples. Looking at the translational semantics, the current work is able to translate the OCL language in LP and to include the semantics of the operations. The difficulties are not in the translation of OCL basic types, predicates, navigations or collections, but merely in the semantics of subtyping or type `OCLAny`. Intuitively this may require the use of order sorted algebra. The semantics of the operations is natural for ADT, it is expressed by axioms. The difficulty comes from "code inheritance". A first approach uses copies of axioms and operator overloading. We also investigate a rewriting system with polymorphic sets of axioms. Formal specification of the meta-level is also a future extension of this work.

References

[ACR94] Pascal André, Dan Chiorean, and Jean-Claude Royer. The Formal Class Model. In *Joint Modular Languages Conference, Modula, Oberon & friends*, ISBN 3-89559-220-X, pages 59–78, Ulm, Germany, 1994. GI, SIG and BCS. September, 28-30.

[Apt90] K. R. Apt. *Logic rogramming*, volume B of *Handbook of Theoretical Computer Science*, chapter 10, pages 495–574. Elsevier, 1990. Jan Van Leeuwen, Editor.

[AR99] Pascal André and Jean-Claude Royer. An Algebraic Approach to the Specification of Heteregeneous Software Systems. In *14th Workshop on Algebraic Development Techniques*, Bonas, France, september 1999. sélectionné pour soumission.

[ARRV00] Pascal André, Annya Romanczuk, Jean-Claude Royer, and Aline Vasconcelos. An Algebraic View of UML Class Diagrams. In *Acte de la conférence LMO'2K*, january 2000.

[BC95] Robert E. Bourdeau and Betty H.C. Cheng. A Formal Semantics for Object Model Diagrams. *IEEE Transactions on Sotfware Engineering*, 21(10):799–821, October 1995.

[BCC+96] Kim Bruce, Luca Cardelli, Guiseppe Castagna, The Hopkins Object Group, Gary T. Leavens, and Benjamin Pierce. On Binary Methods. *Theory and Practice of Object Systems*, 1(3), 1996.

[BHS99] Franck Barbier and Brian Henderson-Sellers. A Survey of the UML's Aggregation and Composition Relationships. *L'objet*, 5(3):21–47, October 1999.

[Cas97] Giuseppe Castagna. *Object-Oriented Programming: A Unified Foundation*. Number ISBN 3-7643-3905-5 in Progress in Theoretical Computer Science. Birkhauser, 1997.

[CLN99] Diego Calvanese, Maurizio Lenzerini, and Daniele Nardi. Unifying class-based representation formalisms. *J. of Artificial Intelligence Research*, 11:199–240, 1999.

[Con99] UML Consortium. The omg unified modeling language specification, version 1.3. Technical report, June 1999. available at ftp://ftp.omg.org/pub/docs/ad/99-06-08.pdf.

[DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. *Rewrite Systems*, volume B of *Handbook of Theoretical Computer Science*, chapter 6, pages 243–320. Elsevier, 1990. Jan Van Leeuwen, Editor.

[dV99] Aline Pires Vieira de Vasconcelos. Formalization of UML Using Algebraic Specifications. Master's thesis, Master EMOOSE, august 1999.

[FBLPS97] Robert France, Jean-Michel Bruel, M. Larrondo-Petrie, and M. Shroff. Exploring the Semantics of UML type structures with Z. In Bowman H. and Derrick J. E, editors, *Proc. 2nd IFIP Workshop on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 247–260. Chapman and Hall, 1997.

[FC99] Liliana Favre and Sylvia Clerici. A Bridge between UML Class Diagrams and Algebraic Specifications. In *14th Workshop on Algebraic Development Techniques*, Bonas, France, September 1999.

[GG89] Stephan Garland and John Guttag. An overview of LP, the Larch Prover. In *Proc. of the third International Conference on Rewriting Techniques and Applications*, volume 355 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.

[GH93] John V. Guttag and James J. Horning, editors. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer Verlag, 1993. With Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing.

[GM87] Joseph A. Goguen and José Meseguer. *Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics*, pages 417–477. Computer Systems Series. Bruce Shriver and Peter Wegner, Research Directions in Object-Oriented Programming, 1987.

[HCRT99] Heinrich Hussmann, Maura Cerioli, Gianna Reggio, and Françoise Tort. Abstract Data Types and UML Models. In *14th Workshop on Algebraic Development Techniques*, Bonas, France, September 1999.

[HHK98a] A. Hamie, J. Howse, and S. Kent. Interpreting the Object Constraint Language. In *Proceedings of Asia Pacific Conference in Software Engineering*. IEEE Press, January 1998.

[HHK98b] A. Hamie, J. Howse, and S. Kent. Modular Semantics for Object-Oriented Models. In *Proceedings of Northern Formal Methods Workshop*, eWics Series. Springer Verlag, August 1998.

[HHK98c] Ali Hamie, John Howse, and Stuart Kent. Navigation expressions in object-oriented modelling. In Egidio Astesiano, editor, *Proceedings Fundamental Approaches to Software Engineering, 1st Internationsl Conference, FASE'98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998*, volume 1382 of *LNCS*, pages 123–150. Springer, 1998.

[Ing86] Daniel H. H. Ingalls. A Simple Technique for Handling Multiple Polymorphism. In Norman Meyrowitz, editor, *Proceedings of the 1986 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 347–349, Portland, OR, october 1986. ACM SIGPLAN.

[KD66] K. Nygaard K. and O-J. Dahl. Simula an Algol-based Simulation Language. *C.ACM*, 9:671–678, September 1966.

[LB98] Kevin Lano and Juan Bicarregui. Semantics and transformations for UML models. In Pierre-Alain Muller and Jean Bézivin, editors, *Proceedings of UML'98 International Workshop, Mulhouse, France, June 3 - 4, 1998*, pages 97–106. ESSAIM, Mulhouse, France, 1998.

[Mey88] Bertrand Meyer. *Object-oriented Software Construction*. International Series in Computer Science. Prentice Hall, 1988.

[Roy92] Jean-Claude Royer. A New Set Interpretation for the Inheritance Relation and its Checking. *ACM OOPS MESSENGER*, 3(3):22–40, july 1992.

[Roy98] Jean-Claude Royer. Type Checking Object-Oriented Programs: Core of the Problem and Some Solutions. *Journal of Object-Oriented Programming*, 11(6):58–66, october 1998. ISSN 0896-8438.

[Roy99] Jean-Claude Royer. UML and ADT: A First Approach to Semantics and Verifications. Research Report 187, Institut de Recherche en Informatique de Nantes, september 1999.

[Wir90] Martin Wirsing. *Algebraic Specification*, volume B of *Handbook of Theoretical Computer Science*, chapter 13, pages 675–788. Elsevier, 1990. Jan Van Leeuwen, Editor.