Universidad Nacional de San Luis

República Argentina

Formal Foundations for Semi-formal Notations

Ana Funes

Asesor: Chris George

Noviembre, 2002

Tesis de Maestría en Ingeniería del Software

Universidad Nacional de San Luis

República Argentina

Ejército de los Andes 950
5700 San Luis
Argentina

Formal Foundations for Semi-formal Notations

Ana Funes
Asesor: Chris George

Abstract

This report presents the semantics for UML class diagrams by using the RAISE Specification Language (RSL) as a formal basis. Property verifications of the model described by the UML class diagram can take place on the corresponding RSL specification by using reasoning techniques supported by the RAISE method. An automated tool that implements the translation and the abstract syntax in RSL for the RSL-translateble class diagrams are also reported.

Tesis de Maestría en Ingeniería del Software

Ana Funes is a teaching assistant at the Department of Computer Science of the National University of San Luis, San Luis, Argentina. Her research interests include formal techniques for Object Oriented development and the use of formal methods for specification. She was a fellow at the International Institute for Software Technologies of the United Nations University (UNU/IIST) in Macau, from September 2001 to May 2002, where she worked on this thesis under the supervision of her adviser Chris George.

Chris George is a Senior Research Fellow at UNU/IIST, 1 September 1994 - 31 August 2003. He is one of the main contributors to RAISE, particularly the RAISE method, and that remains his main research interest. Before coming to UNU/IIST he worked for companies in the UK and Denmark.

Contents

List of Figures iii

1 Introduction 1

2 Class Diagrams and UML 3
- 2.1 The Domain Model . . . 3
- 2.2 The Design Class Diagram . . . 5

3 Syntactic and semantic reference 7
- 3.1 RSL and the RAISE method . . . 7

4 Formal Syntax 9

5 Formal Semantics 24
- 5.1 The Class . . . 24
- 5.2 Relationships . . . 28
  - 5.2.1 Association . . . 28
  - 5.2.2 Generalization . . . 33
  - 5.2.3 Dependency . . . 39
- 5.3 Class Diagrams . . . 39
- 5.4 Initial value of an attribute . . . 41
- 5.5 Association Navigation . . . 41
- 5.6 Composition and Aggregation . . . 43
- 5.7 Parameterized Classes . . . 44
- 5.8 Constraints . . . 50
  - 5.8.1 Multiplicities . . . 51
  - 5.8.2 Attribute and Operation Scope . . . 53
  - 5.8.3 Attribute and Association End Properties . . . 56
  - 5.8.4 Abstract Classes and Abstract Operations . . . 57
  - 5.8.5 Existence and Bi-navigation constraints . . . 58

6 The Tool 61

7 Concluding Remarks 63

Acknowledgments 65

Appendices 66

A Templates for a Class 67

B Templates for a Subclass 70

C Templates for Leaf Classes 73

D Template for a Class Diagram **75**

E Templates for Parameterized Classes **78**
E.1 Template for Template Classes 78
E.2 Templates for Instantiated Classes 79

F Templates for constraints **81**
F.1 Class multiplicities 81
F.2 Association end multiplicities 81
F.3 Attribute multiplicities 82
F.4 Class-scope attributes 84
F.5 Attribute and association-end properties 84
F.6 Abstract classes 85
F.7 Existence constraints 85
F.8 Bi-navigability constraints 86

G RSL Specification - Examples **87**
G.1 RSL Specification for the example of domain class diagram (figure 1) 87

List of Figures

1 Domain model for a Point of Sale System 4
2 RSL Module Dependency Graph 5
3 Example of design class diagram 6
4 Example of a class in UML 25
5 An UML class diagram 29
6 An association class 31
7 Decomposition of an association class 31
8 Decomposition of an association class 32
9 Example of N-ary association 32
10 Decomposition of N-ary association 32
11 Example of Generalization in UML 33
12 RSL module dependency graph 37
13 Multiple inheritance 37
14 Delegation 38
15 Association Navigation in N-ary associations 42
16 Composition 43
17 Template and Instantited Classes 45
18 Inheritance plus Template Classes 49
19 Multiplicities 51
20 Scopes for attributes and operations 54
21 Example of abstract class 57

22 Component diagram 61

1 Introduction

Requirements analysis is usually the first phase in software development. In this stage, it is necessary to specify the exact requirements of the system to be developed and, for this reason, it is crucial for the quality of the product that a high degree of interaction between the customer, the end users and the developer takes place.

An important technique for increasing reliability of software is the use of formal specifications, that is, the representation of software by means of formal notations -a language with a precise syntax, a precise semantics and a proof system. The use of formal specifications has benefits ranging from the possibility of building unambiguous specifications, to the possibility of proving system properties, to automatic code generation. However, since they require a high level of expertise in algebra and mathematical logic, they make the communication with the end users to validate requirements difficult.

On the other hand, the use of graphical notations has shown itself to be useful when interacting with the end users. Many engineers customarily adopt pictorial notations for their blueprints since graphical descriptions can be more intuitive and easier to grasp than textual descriptions –according to folk wisdom, a picture is worth a thousand words. Many forms of more-or-less formal diagrams have been used in Software Engineering for a long time. Many practitioners have adopted the use of graphical notations such as Data Flow diagrams [8], Entity-Relationship diagrams [6], State Charts [12] and process-oriented approaches like Jackson System Development [14] [26]. However, lately, object-oriented approaches seems to be the most popular. Object orientation has evolved from a programming paradigm to methods that cover the complete life cycle. There are many variations of these methods such us OMT [23], OOSE [15], Booch methods [3] [4] and the Unified Software Development Process [24] which has unified the notation used by the previous through the use of the Unified Modeling Language (UML) [5].

UML is a graphical language for building object-oriented models of software systems. Nowadays, it has become the de facto standard for object oriented modeling, and there exists a wide variety of graphical tools that support it. However, although UML notations are easily communicated, their semantics are informal and –consequently– they can be ambiguous, leading to misunderstandings. Here, Formal Specifications can play an important role. They can be used as the formal foundations for expressing the semantics of the semi-formal language.

Having this formal semantics as basis, a combination of graphical and formal notations can be achieved in order to overcome the problems present in both formal and semi-formal specifications without losing their respective benefits, that is, the understandability of graphical notations and the unambiguity of formal specifications.

Another important consequence of having a formal representation of an informal or semi-formal model is the possibility of reasoning about their properties.

Several studies have been done using formal techniques for expressing the semantics of object-

oriented models (see [9] [7] [20] [16]). In this work we explore the use of RSL [10] –the language of the RAISE development method [11]– to give the formal foundations for UML class diagrams. An automated tool to perform the translation is also presented.

The general structure of this report is as follows. Section 2 introduces class diagrams in UML and their use in different stages of the software development process. Section 3 introduces the concepts of semantic and syntactic reference, and gives a concise description of RSL and the RAISE method. In section 4, the formal syntax in RSL for UML class diagrams is given. Section 5 presents the formal semantics in RSL for class diagrams. Section 6 describes briefly the tool developed to achieve the translation, and finally Section 7 concludes and discusses future work.

2 Class Diagrams and UML

UML is a graphical language developed to build different kind of diagrams, which can be used in different phases of the software development process. Since it is only a language, it is just a tool for the development. This means it is process independent, and consequently, it does not tell you what models you should create and when you should create them, but it only provides the tools to build them.

Several kind of diagrams can be created using UML in order to cover different views of a system. In this work we are only concerned with class diagrams.

Class diagrams provide a static view of the modeled system. Sometimes they are used for modeling the vocabulary of the system. This implies a decision on which concepts or entities are part of the system and which are outside its boundaries. They are also used to build domain models, where all the concepts present in the application domain are shown in the diagram, as well as the relations between them. They can be used also to model collaborations among a set of classes, which work together to provide a collaborative behavior, or even to represent a database schema, among others.

It is possible to build class diagrams at different abstraction levels and with different degrees of detail. For instance, conceptual models –which are typically used in the first phase of the development– have no implementation details, while design class diagrams should. In the next two sections we discuss conceptual models and design class models and give examples of both.

2.1 The Domain Model

A domain model –often referred to as a conceptual model– might be represented by a particular kind of UML class diagram. This model explains the structure of the application domain rather than the application structure itself. It focuses on the domain concepts, not on the software entities. Therefore, although most of their elements will be present in the design model later on, some of them will not, and new ones could even appear.

This kind of model can be represented by UML class diagrams that commonly consist only of classes and relationships between them. The classes, which represent the identified concepts in the domain, only have some attributes. Operations should not be present. Here, the most frequent relationship between classes is the Association, which may have adornments on its ends. An adornment may be a role name, a multiplicity, an aggregation and/or composition adornment.

Another possible relationship among classes sometimes used in this kind of model is Generalization. Generalization is an inheritance relation among two or more classes, sometimes called "is-a-kind-of".

In figure 1 we give an example taken from [17] of a conceptual model in UML built for a simple system: a Point of Sale System. The model only considers the concepts and associations identified for the use cases "Buy an item with cash" and "Initiate the system". On the analysis of the remaining discovered use cases, the domain model may be extended with new classes, attributes and associations.

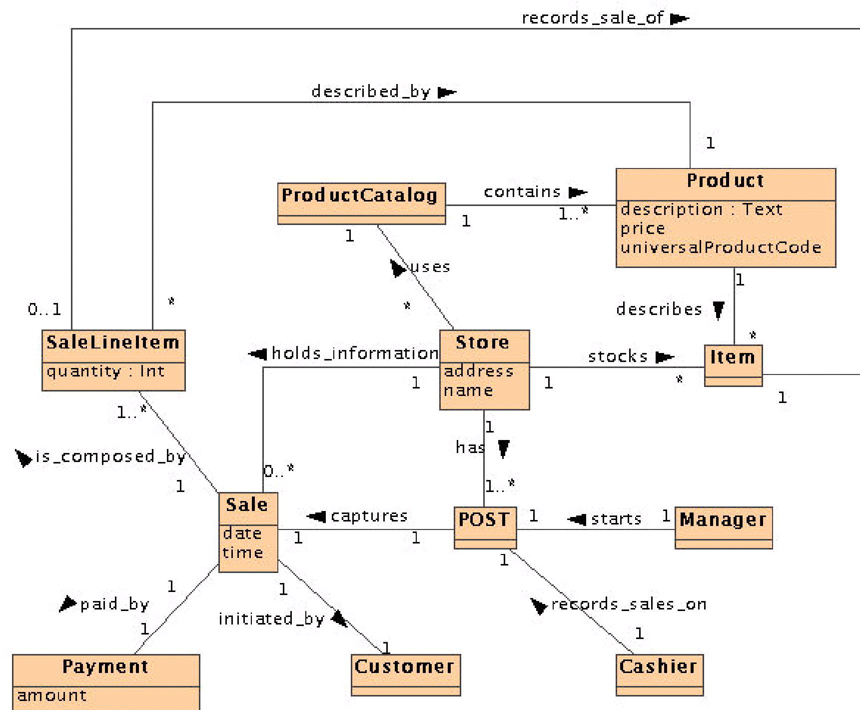

Figure 1: Domain model for a Point of Sale System

In section 5 we present the semantics for all the basic constructors used in a class diagram as well as for some other language features used mainly in design class diagrams. According to these semantics an RSL specification for the conceptual model can be derived. The resulting RSL specification is modular. For each class present in the class diagram, two RSL modules are created. One has the specification for an object of the class, and the other, which uses the former, has the specification for the class. Those modules that have the specification for an object of the class use, in turn, an RSL module named "TYPES", which has all the definitions for the abstract types in the model. The whole specification consists of a top level module "S", which uses the modules containing the specification of each class. For the previous example, we have "S" using 'SALELINEITEMS", "MANAGERS", "PRODUCTS", "SALES", "PRODUCTCATALOGS", "PAYMENTS", "'POSTS", "CASHIERS", "ITEMS", "CUSTOMERS", and "STORES", each one corresponding to a class in the class diagram. These use respectivelly "SALELINEITEM", "MANAGER", "PRODUCT", "SALE", "PRODUCTCATALOG", "PAYMENT", "'POST", "CASHIER", "ITEM", "CUSTOMER", and "STORE". Finally, the last use, in turn, the module "TYPES". Figure 2 below shows the dependency module graph

produced by the RSL tool for the specification obtained from the class diagram.

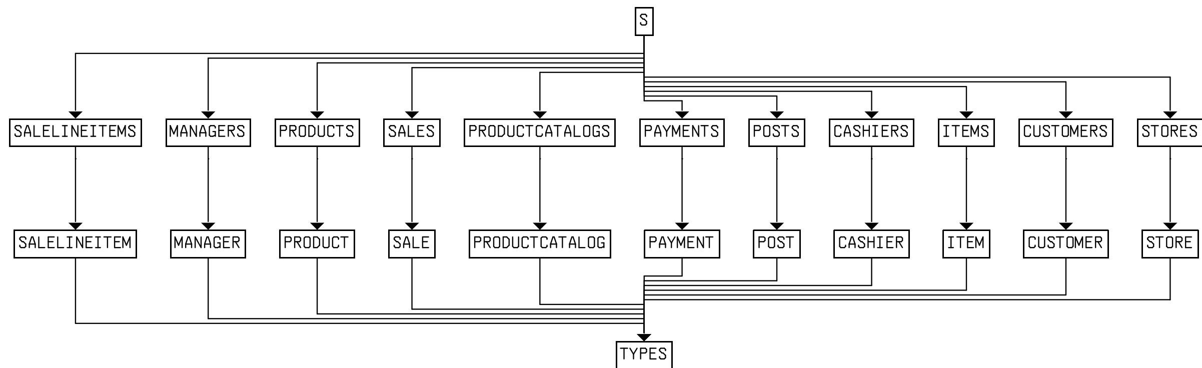

Figure 2: RSL Module Dependency Graph

Domain models usually have properties that must hold. These properties may be expressed as UML constraints in the class diagram. UML has no restriction for expressing constraints, which means they can be written in natural language or using any other language. Therefore, we have the possibility of expressing in RSL all these properties without any ambiguity on the basis of the specification obtained from the class diagram.

The corresponding RSL specification for the class diagram shown in figure 1 can be found in Appendix G.1 on page 87.

2.2 The Design Class Diagram

This kind of class diagram reflects a solution-oriented structure, while the domain model is problem-oriented. That is, design class diagrams are concerned with the way in which the solution is given, and conceptual models with the entities and the relationships present in the problem domain.

The design class diagram can be obtained after an analysis of interaction and collaboration between objects has taken place. Besides the structural elements present in domain models, design class diagrams have other details that can be expressed in UML such as all the methods identified in the collaboration diagrams, navigation in the association ends, scope and type of the attributes and operations, and even new associations discovered during the design phase. Not all the classes present in the conceptual model will be part of the design diagram of the system, but only those that participate in the object interactions in order to achieve the functionality required for the software system.

An example of a partial Design class diagram for a Point of Sale System with only one POST is depicted in figure 3. From this semi-formal model, we can obtain an RSL formal specification as well.

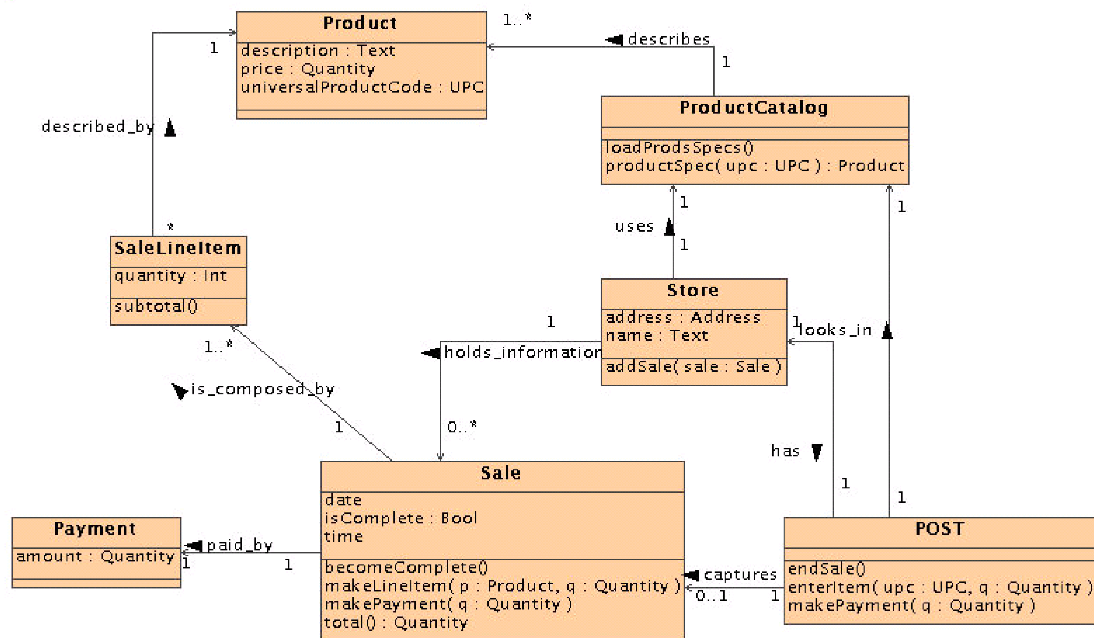

Figure 3: Example of design class diagram

From both domain models and design models, we can derive RSL specifications. These specifications can be used to carry out rigorous analysis of the properties of the model represented by the class diagram or on new properties expressed in RSL on the basis of the obtained specification. These specifications can be used as well as initial specifications to be further refined to finally be translated to executable code.

3 Syntactic and semantic reference

When a language is used as a reference for another one, the first one can be used to express the syntax and/or semantics of the second one. That is, the allowed elements and their formation rules (abstract syntax) of a given language, and the properties of the syntactical elements are described using a more abstract representation in the other language.

UML is a graphical language that contains a set of elements and formation rules for building diagrams. Using UML it is possible to build different models for each phase of the software development process either following a particular development method or just for drawing diagrams used as documentation.

Although many graphical tools are available to build syntactically correct UML diagrams (see `http://www.jeckle.de/umltools.htm`), we are interested, in this case, in the use of UML for building a particular kind of class diagram. Therefore, whether an ad-hoc graphical tool is built or an existing one is adopted, the syntax for the class diagrams should be given. In the first case, the syntax is used by the tool for building syntactically correct class diagrams. In the second case, when an available working UML tool is used, the syntax guides the parsing process for the outputs generated by the tool.

In the present work, both abstract syntax and semantics of the class diagrams are given in terms of RSL, i.e. RSL becomes a meta-language in which the semantics and abstract syntax of UML class diagrams are defined.

3.1 RSL and the RAISE method

RSL (**R**AISE **S**pecification **L**anguage) is a formal specification language, which receives its name from the RAISE method. The RAISE (**R**igorous **A**pproach to **I**ndustrial **S**oftware **E**ngineering) method was the result of an European collaborative project carried out between the years 1985 and 1990 in the context of the CEC ESPRIT programme. The project, which not only developed the method and the specification language, gave as result also a comprehensive set of tools. UNU/IIST has produced a portable type checker for RSL, and a number of associated tools: a pretty printer, a "confidence condition" generator, a translator to Standard ML, and a translator to C++. All of them are freely available and can be downloaded from the UNU/IIST ftp site: ftp.iist.unu.edu, in /pub/RAISE/rsltc by logging in as "anonymous".

The method consists of a number of techniques and strategies for doing formal development and proofs. Its language, RSL, is a wide spectrum specification language. It allows the use of different styles of specifications: applicative or imperative; sequential or concurrent; direct (explicit) or axiomatic (implicit); algebraic (with abstract data types) or model-oriented (with concrete data types). More information about RSL and the RAISE Method can be found in [10] [11].

In the present work, the RSL specifications produced as the result of the semantic analysis of UML class diagrams are written using an applicative and sequential style.

4 Formal Syntax

As we have discussed in section 2, class diagrams can be used for modeling the concepts and interactions in the application domain as well as to create software system models, as in the case of design class diagrams.

In this section we give the abstract syntax for UML class diagrams but we do not enter into details about the meaning of each syntactic element or when they are used. In section 5, for each syntactic element, we explain in the first place its meaning based on UML documents [5] [22], and show some examples, then we give the corresponding semantics in RSL.

We give the syntax in a top-down fashion, starting by the class diagram. A class diagram, in UML, is formed by classes and the relationships among them. In order to build a well-formed class diagram a set of rules must be observed on the classes, the relationships and on the whole class diagram.

A class diagram is just a well-formed pair of well-formed classes and well-formed relationships.

```
type
   ClassDiagram =
      {| cd : ClassDiagram1 • well_formed(cd) |},
   ClassDiagram1 ::
      classes : Class-set
      rels : Rel-set
```

Before specifying what a well-formed class means, we give its structure. A class consists of a name, a set of attributes, a set of operations and a multiplicity. It can be abstract, root, leaf and can even have parameters in the case of the template classes.

```
type
   Class = {| c : Class1 • well_formed(c) |},

   Class1 ::
      name : Name
      attributes : Attribute-set
      operations : Operation-set
      multiplicity : Multiplicity
      is_abstract : Bool
      is_root : Bool
      is_leaf : Bool
      parameters : FormalParameter*,
```

Name = **Text**,

Multiplicity :: lower : Bound upper : Bound,

Bound == a_Nat(n : **Nat**) | asterisk,

Each attribute has a name, an optional type, a multiplicity, a scope (classifier or instance), and a changeability (frozen, add only or changeable).

type
Attribute ::
name : Name
at_type : Op_Type
multiplicity : At_multiplicity
scope : Scope
changeability : Changeability,

Op_Type == null | a_Type(typ : Type),

Type = **Text**,

At_multiplicity ::
lower : At_bound
upper : At_bound,

At_bound ==
a_Nat(n : **Nat**) | asterisk | a_parameter(name : Name),

Scope == classifier | instance,

Changeability == frozen | addonly | changeable

An operation consists of a name, a list of formal parameters, where each parameter has a name and an optional type. Furthermore, an operation has an optional result-type, a scope and can be abstract.

type
Operation ::
name : Name
parameters : FormalParameter*
result : Op_Type
is_abstract : **Bool**

```
    scope : Scope,

  FormalParameter ::
    name : Name
    par_type : Op_Type
```

A well-formed class must satisfy a set of properties, that we express as predicates in the function "well_formed" used to define the type "Class". Following, we give all these properties:

- A well formed class must not have duplicated attribute names.

```
∀ at1, at2 : Attribute •
    (at1 ∈ attributes(c) ∧
      at2 ∈ attributes(c) ∧ at1 ≠ at2) ⇒
        name(at1) ≠ name(at2)
```

- The multiplicity of each attribute must be well formed.

```
∀ at : Attribute •
    at ∈ attributes(c) ⇒
        wf_AttMultiplicity(
          lower(multiplicity(at)),
           upper(multiplicity(at)), c)
```

- Changeability "addOnly" can be used only in attributes whose multiplicities are not a fixed number(see [24], page 185) and are greater that one (see [5]).

```
∀ at: Attribute •
    changeability(at) = addonly ⇒
        case lower(multiplicity(at)) of
          a_Nat(l) →
            case upper(multiplicity(at)) of
              a_Nat(u) → l ≠ u ∧ u > 1,
              _ → true
            end,
          _ → true
        end
```

- Duplicated operation names are accepted only when they differ in scope or in types of their parameters or result value.

$$
\begin{array}{l}
\forall\ o1, o2 : \text{Operation} \bullet \\
\quad (o1 \in \text{operations}(c) \land \\
\quad o2 \in \text{operations}(c) \land \text{name}(o1) = \text{name}(o2) \land \\
\quad o1 \neq o2) \Rightarrow \\
\qquad (\text{scope}(o1) \neq \text{scope}(o2) \lor \\
\qquad \text{result}(o1) \neq \text{result}(o2) \lor \\
\qquad \text{different_types}(\text{parameters}(o1), \text{parameters}(o2)))
\end{array}
$$

- If a class has an abstract operation, it must be abstract.

$$
\begin{array}{l}
(\exists\ o : \text{Operation} \bullet \\
\quad o \in \text{operations}(c) \land \text{is_abstract}(o)) \Rightarrow \\
\text{is_abstract}(c)
\end{array}
$$

- Abstract classes have at least one abstract operation.

$$
\begin{array}{l}
\text{is_abstract}(c) \Rightarrow \\
\quad (\exists\ o : \text{Operation} \bullet \\
\qquad o \in \text{operations}(c) \land \text{is_abstract}(o))
\end{array}
$$

- The multiplicity of the class must be valid, it means an order between the lower bound and the upper bound must be observed.

$$
\text{lower}(\text{multiplicity}(c)) \leq \text{upper}(\text{multiplicity}(c))
$$

- There are no duplicated parameter names in a template class.

$$
\begin{array}{l}
\forall\ p1, p2\text{: FormalParameter} \bullet \\
\quad (p1 \in \text{parameters}(c) \land p2 \in \text{parameters}(c) \land p1 \neq p2) \\
\qquad \Rightarrow \text{name}(p1) \neq \text{name}(p2)
\end{array}
$$

- Since an abstract class is an incomplete class because it has abstract methods, i.e., methods without an implementation, it must be extended (inherited) in the class diagram by at least one concrete subclass. Therefore it cannot be a leaf.

$$
\text{is_abstract}(c) \Rightarrow \sim\text{is_leaf}(c)
$$

Some auxiliary functions must be defined in order to complete the specification for the syntax of a class.

```
/* Checks if multiplicity bounds are well−formed */
≤ : Bound × Bound → Bool
lo ≤ up ≡
   case lo of
      a_Nat(l) →
         case up of
            a_Nat(u) → l ≤ u,
            asterisk → true
         end,
      asterisk →
         case up of
            a_Nat(u) → false,
            asterisk → true
         end
   end,

/* Checks if an attribute  multiplicity is well−formed */
wf_AttMultiplicity :
   At_bound × At_bound × Class → Bool
wf_AttMultiplicity(lo, up, c) ≡
   case lo of
      a_Nat(l) →
         case up of
            a_Nat(u) → l ≤ u,
            asterisk → true,
            a_parameter(name) → isin_formalPars(name, c)
         end,
      asterisk →
         case up of
            asterisk → true,
            _ → false
         end,
      a_parameter(name) →
         if ∼(isin_formalPars(name, c)) then false
         else
            case up of
               a_Nat(u) → true,
               asterisk → true,
               a_parameter(name) →
                  isin_formalPars(name, c)
            end
         end
   end,

/* Informs if a given name is a formal parameter of
```

```
a given class c */
isin_formalPars : Name × Class → Bool
isin_formalPars(name, c) ≡
   (∃ i : Int •
      i ∈ inds parameters(c) ∧
      name(parameters(c)(i)) = name),

/* Informs if an attribute multiplicity bound
is a parameter */
is_parameterized : At_bound → Bool
is_parameterized(b) ≡
   case b of
      a_parameter(p) → true,
      _ → false
   end,

/* Informs if two parameter lists have different
types */
different_types :
   FormalParameter* × FormalParameter* → Bool
different_types(pl1, pl2) ≡
   len pl1 = len pl2 ⇒
      (∃ i : Int •
         i ∈ inds pl1 ∧
         par_type(pl1(i)) ≠ par_type(pl2(i)))
```

In a class diagram, the classes can be related through different kind of relationships. Basically, they are classified into three types: associations, generalizations and dependencies. Instantiations are viewed in UML as stereotyped dependencies, but since each stereotyped element has a particular meaning and we are interested specifically in instantiation, we separate it from general dependencies.

Each relationship in a class diagram must be well formed. Therefore, we define the structure for each type of relationship and we give the well formedness rules, as we did for a class.

```
type
   Rel = {| r : Rel1 • well_formed(r) |},

   Rel1 =
      Association | Generalization | Dependency |
      Instantiation
```

An association is a relationship among n classes. Each association end holds information not

only about the class but it has several adornments: a multiplicity, a navigability, it can be composite or aggregate and –like attributes– it has a given changeabilty. Furthermore, each association in the diagram has an assigned name.

```
type
   Association ::
      name : Name
      ends : End-set,

   End ::
      end_class : Class
      multiplicity : Multiplicity
      navigable : Bool
      kind : EKind
      changeability : Changeability,

   EKind == composite | aggregate | none
```

Generalizations, dependencies, as well as instantiations, are all binary relationships. In the case of instantiations, besides the data about the involved classes, it is necessary to hold all the information about the actual parameters.

```
type
   Generalization ::
      subclass : Class
      superclass : Class,

   Dependency ::
      source : Class
      target : Class,

   Instantiation ::
      template : Class
      instantiated : Class
      actualparameters : Value*,

   Value
```

Below we give the boolean function "well_formed" used to define well-formed relationships. Each kind of relation have different properties.

value

```
well_formed : Rel1 → Bool
well_formed(r) ≡
  case r of
    mk_Association(_, ends) → well_formedAsso(ends),
    mk_Generalization(subclass, superclass) →
      well_formedGen(subclass, superclass),
    mk_Instantiation(
       template, instantiated, actualParameters) →
      well_formedIns(
          template, instantiated, actualParameters),
    _ → true
  end
```

When the relationship is an association a series of constraints on the association ends are imposed. All of them are defined in the function "well_formedAsso" used by "well_formed".

- An association is an N-ary relationship with N ≥ 2.

```
card ends ≥ 2
```

- The multiplicities of the association ends must be well formed, that is, the multiplicity lower bound must be less or equal to multiplicity upper bound.

```
∀ e: End •
    (e ∈ ends ⇒
        lower(multiplicity(e)) ≤
            upper(multiplicity(e)))
```

- Changeability "addOnly" can be used only in association ends whose multiplicities are not a fixed number (see [24], page 185) and are greater that one (see [5]).

```
∀ e: End •
    changeability(e) = addonly ⇒
        case lower(multiplicity(e)) of
          a_Nat(l) →
            case upper(multiplicity(e)) of
              a_Nat(u) → l ≠ u ∧ u > 1,
              _ → true
            end,
          _ → true
        end
```

- When the association has one aggregate or composite end, the association must be binary.

$$
\begin{array}{l}
\forall \text{ e: End} \bullet \\
\quad ((\text{e} \in \text{ends} \wedge \\
\quad\quad (\text{kind(e)} = \text{composite} \vee \text{kind(e)} = \text{aggregate})) \Rightarrow \\
\quad\quad\quad \textbf{card} \text{ ends} = 2)
\end{array}
$$

- In a composite association (or just composition), the parts of a composition must be navigable.

$$
\begin{array}{l}
\forall \text{ e1, e2: End} \bullet \\
\quad ((\text{e1} \in \text{ends} \wedge \text{kind(e1)} = \text{composite} \wedge \\
\quad\quad \text{e2} \in \text{ends} \wedge \text{e2} \neq \text{e1}) \Rightarrow \text{navigable(e2)})
\end{array}
$$

- Only one end can be aggregated or composite, i.e. just one can play the role of "whole".

$$
\begin{array}{l}
\forall \text{ e: End} \bullet \\
\quad ((\text{e} \in \text{ends} \wedge \\
\quad\quad (\text{kind(e)} = \text{composite} \vee \text{kind(e)} = \text{aggregate})) \Rightarrow \\
\quad\quad\quad \sim(\exists \text{ e1 : End} \bullet \\
\quad\quad\quad\quad (\text{e1} \in \text{ends} \wedge \text{e1} \neq \text{e} \wedge \\
\quad\quad\quad\quad (\text{kind(e1)} = \text{composite} \vee \\
\quad\quad\quad\quad\quad \text{kind(e1)} = \text{aggregate}))))
\end{array}
$$

- In a composition, the end corresponding to the "whole" cannot have a multiplicity greater than 1.

$$
\begin{array}{l}
\forall \text{ e: End} \bullet \\
\quad ((\text{e} \in \text{ends} \wedge \text{kind(e)} = \text{composite}) \Rightarrow \\
\quad\quad\quad \text{upper(multiplicity(e))} \leq \text{a_Nat(1)})
\end{array}
$$

- In an association, at least one of the ends must be navigable.

$$
\exists \text{ e : End} \bullet (\text{e} \in \text{ends} \wedge \text{navigable(e)})
$$

When there is inheritance between two classes, the generalization relationship must satisfy three constraints: a subclass cannot be root; the superclasses cannot be leaf and the subclasses cannot redefine the attributes of their superclasses.

```
well_formedGen : Class × Class → Bool
well_formedGen(subclass, superclass) ≡

    /* A subclass cannot be a root class.*/
    ~is_root(subclass) ∧

    /* A superclass cannot be a leaf class.*/
    ~is_leaf(superclass) ∧

    /* A subclass cannot redefine the attributes of
    the superclass */
    (∀ at1, at2 : Attribute •
        (at1 ∈ attributes(subclass) ∧
          at2 ∈ attributes(superclass)) ⇒
           name(at1) ≠ name(at2))
```

When the relationship is an instantiation, the instantiated class cannot extend or redefine the structure and behavior of the template class, which means that the instantiated class is fully specified by the template. Moreover, the class that plays the role of template in the relationship must have at least one formal parameter, and the actual parameters of the instantiation must match the formal parameters of the template class.

```
well_formedIns : Class × Class × Value* → Bool
well_formedIns(
     template, instantiated, actualParameters) ≡

    /* An instantiated class is fully specified
    by its template. */
    (card attributes(instantiated) = 0 ∧
     card operations(instantiated) = 0 ∧
     multiplicity(instantiated) = multiplicity(template) ∧
     is_abstract(instantiated) = is_abstract(template) ∧
     is_leaf(instantiated) = is_leaf(template) ∧
     is_root(instantiated) = is_root(template)  ∧

    /* A template class must have at least one formal
    parameter */
    (len parameters(template) > 0) ∧

    /* The actual parameters must match with the formal
    parameters in the template */
     match_ps(parameters(template), actualParameters))
```

Having specified what well-formed classes and well-formed relations are, now we can give the properties that must hold when a well-formed class diagram is being built.

- In a class diagram, all the classes that are involved in a relationship must belong to the set of classes in the class diagram.

```
∀ rel : Rel •
    (rel ∈ rels(cd) ⇒
        case rel of
          mk_Instantiation(template, instantiated, _) →
            template ∈ classes(cd) ∧
            instantiated ∈ classes(cd),
          mk_Dependency(source, target) →
            source ∈ classes(cd) ∧
            target ∈ classes(cd),
          mk_Generalization(subclass, superclass) →
            subclass ∈ classes(cd) ∧
            superclass ∈ classes(cd),
          mk_Association(_, ends) →
            (∀ e: End • (e ∈ ends ⇒
              end_class(e) ∈ classes(cd)))
        end)
```

- Each class must have a unique name, i.e. there are no duplicated class names.

```
∀ c1, c2 : Class •
    ((c1 ∈ classes(cd) ∧ c2 ∈ classes(cd) ∧
      c1 ≠ c2) ⇒ name(c1) ≠ name(c2))
```

- UML allows the use of rolenames at the association ends. This allows to have different associations with the same name relating a common class, if they differ in the rolenames. However, as we do not use rolenames in our class diagrams, a well-formed class diagram cannot have two different associations with the same name relating a common class.

```
∀ a1, a2 : Association •
    ((a1 ∈ rels(cd) ∧ a2 ∈ rels(cd) ∧
      a1 ≠ a2 ∧ name(a1) = name(a2)) ⇒
        end_classes(a1) ∩ end_classes(a2) = {})
```

- When in a class diagram there is an abstract class, it must be the superclass of at least one class in the class diagram in order to assure the existence of a concrete subclass in the inheritance hierarchy that make it useful.

$$
\begin{array}{l}
\forall\ c\text{: Class} \bullet \\
\qquad ((c1 \in \text{classes}(cd) \wedge \text{is_abstract}(c1)) \Rightarrow \\
\qquad\quad (\exists\ g : \text{Generalization} \bullet \\
\qquad\qquad g \in \text{rels}(cd) \wedge \text{superclass}(g) = c1))
\end{array}
$$

- Since a template class cannot be used directly because it has unbound parameters, it cannot be either the target of an association or a superclass (see [22], page 52, section 3).

$$
\begin{array}{l}
(\forall\ i\text{: Instantiation, } a\text{: Association, } e\text{: End} \bullet \\
\qquad ((i \in \text{rels}(cd) \wedge a \in \text{rels}(cd) \wedge \\
\qquad\quad e \in \text{ends}(a) \wedge \\
\qquad\quad \text{end_class}(e) = \text{template}(i)) \Rightarrow \sim\text{navigable}(e))) \wedge \\
(\forall\ i\text{: Instantiation, } g\text{: Generalization} \bullet \\
\qquad ((i \in \text{rels}(cd) \wedge g \in \text{rels}(cd)) \Rightarrow \\
\qquad\qquad \text{template}(i) \neq \text{superclass}(g)))
\end{array}
$$

- As an instantiated class is fully specified by its template, it cannot be the source of an association. It can only be a target in an association.

$$
\begin{array}{l}
(\forall\ i\text{: Instantiation, } a\text{: Association, } e\text{: End} \bullet \\
\qquad ((i \in \text{rels}(cd) \wedge a \in \text{rels}(cd) \wedge \\
\qquad\quad e \in \text{ends}(a) \wedge \\
\qquad\quad \text{end_class}(e) = \text{instantiated}(i)) \Rightarrow \\
\qquad\qquad (\forall\ e2 : \text{End} \bullet \\
\qquad\qquad\quad (e2 \in \text{ends}(a) \wedge e2 \neq e) \Rightarrow \\
\qquad\qquad\qquad \sim\text{navigable}(e2))))
\end{array}
$$

- Although an instantiated class is fully specified by its template, it can be a subclass (see [22], page 55, section 3), however we do not allow the use of instantiated classes that are also subclasses due the semantics given for inheritance (see section 5.2.2).

$$
\begin{array}{l}
(\forall\ i\text{: Instantiation, } g\text{: Generalization} \bullet \\
\qquad ((i \in \text{rels}(cd) \wedge g \in \text{rels}(cd)) \Rightarrow \\
\qquad\qquad \text{subclass}(g) \neq \text{instantiated}(i)))
\end{array}
$$

- An instantiated class is the result of the instantiation of only one template, that means the same class cannot appear as instantiated in two different instantiation relationships.

$$
\begin{array}{l}
\forall\ i1, i2\text{: Instantiation} \bullet \\
\qquad ((i1 \in \text{rels}(cd) \wedge i2 \in \text{rels}(cd) \wedge \\
\qquad\quad i1 \neq i2) \Rightarrow \\
\qquad\qquad \text{instantiated}(i1) \neq \text{instantiated}(i2))
\end{array}
$$

- When two classes are related by an instantiation, no other relationship between them is possible, which means that no association, generalization or a different instantiation between the involved classes can take place. In the predicate below, we do not check the existence of another instantiation between the classes because it is given by the two previous properties which establish that a class cannot be instantiated by more than one template and, a template cannot be an instantiated class.

```
∀ i: Instantiation, a: Association, e: End ,
      g: Generalization, at: Attribute •
      (i ∈ rels(cd) ⇒
            ((a ∈ rels(cd) ⇒
                  ∼there_is_association(
                        template(i), instantiated(i), a)) ∧
             ((g ∈ rels(cd) ∧
                subclass(g) = template(i)) ⇒
                  superclass(g) ≠ instantiated(i)) ∧
             (at ∈ attributes(template(i)) ⇒
                  at_type(at) ≠
                     a_Type(name(instantiated(i))))))
```

- When two classes are related by a generalization, the only allowed relationships between them are associations. It is not necessary to check absence of instantiation because it is given by the previous predicates. Neither to check absence of generalization on the same classes in the inverse order because it is given by the next property which establishes that generalization has no loops. We do not give either the property that establishes that it is not possible to inherit more than one time from the same class because can be inferred from the property that establishes that multiple inheritance is not allowed.
- In a class diagram there are no loops among generalizations.

```
let
   gs = gens(cd) ∪ inss(cd),
   trans = closure(gs)
in
   ∼existsRefSymmPairs(gs)
end
```

- Although UML allows the use of multiple inheritance, we have restricted the well-formed diagrams to not allow multiple inheritance. Note that the predicate below prevents inheriting more than once.

```
∀ g1, g2: Generalization •
      ((g1 ∈ rels(cd) ∧ g2 ∈ rels(cd) ∧
         g1 ≠ g2) ⇒ subclass(g1) ≠ subclass(g2))
```

In order to complete the specification for a well-formed class diagram, following we give the auxiliary functions used before.

```
value
   /* Return the set of classes obtained from the set of ends of a given association */
    end_classes : Association → Class-set
    end_classes(a) ≡ {end_class(e) | e : End • e ∈ ends(a)},

   /* Return the set of pairs (subclass, superclass) related by a
    generalization of degree one in a class diagram */
   gens : ClassDiagram → (Class × Class)-set
   gens(cd) ≡
      {(c1, c2) |
       c1, c2 : Class •
         (∃ g : Generalization •
               g ∈ rels(cd) ∧
               (c1, c2) = (subclass(g), superclass(g)))},

   /* Return the set of pairs (instantiated, template) related by an
    instantiation in the class diagram */
   inss : ClassDiagram → (Class × Class)-set
   inss(cd) ≡
      {(c1, c2) |
       c1, c2 : Class •
         (∃ i : Instantiation •
               i ∈ rels(cd) ∧
               (c1, c2) = (instantiated(i), template(i)))},

   /* Returns the transitive closure of a relation r */
   closure : (Class × Class)-set → (Class × Class)-set
   closure(r) ≡
      {(c1, c2) |
       (c1, c2) : Class × Class •
         (c1, c2) ∈ r ∨
         (∃ c : Class •
               (c1, c) ∈ r ∧ (c, c2) ∈ closure(r))},

   /* Inform if two classes c1 and c2 are involved in the association a */
   there_is_association :
      Class × Class × Association → Bool
   there_is_association(c1, c2, a) ≡
      (∃ e1, e2 : End •
         e1 ∈ ends(a) ∧ e2 ∈ ends(a) ∧ e1 ≠ e2 ∧
         c1 = end_class(e1) ∧ c2 = end_class(e2)),
```

```
/* Inform if c1 is a subclass of c2 because of
 a direct or transitive generalization in cd */
there_is_generalization :
   Class × Class × ClassDiagram → Bool
there_is_generalization(c1, c2, cd) ≡
   let gs = gens(cd) ∪ inss(cd), g = closure(gs) in
      (c1, c2) ∈ g
   end,

/* Inform if a given relation r has symmetric
or reflexive pairs */
existsRefSymmPairs : (Class × Class)-set → Bool
existsRefSymmPairs(r) ≡
   (∃ (c1, c2) : Class × Class •
       (c1, c2) ∈ r ∧ (c2, c1) ∈ r)
```

5 Formal Semantics

To give the semantics of a language in terms of another, it is necessary to define a mapping from the source language to the language used as reference. Also a mechanism, which implements the translation should be provided. In our case, this mechanism should be able to take an abstract representation built from a syntactically correct UML class diagram and produce as output an RSL specification.

The process applied to explore the semantic foundations of UML class diagrams has given as result a set of RSL-templates that are described in the appendices. These templates are the basis for the automatic translation mechanism proposed, where notations in UML are directly translated into an RSL specification, which does not refer explicitly to notations of UML (like class, association, etc.) but defines project-specific instances for them.

We first proceed by analyzing informally, through examples in UML, the semantics of a given element. Then, we give formally these semantics in RSL. From these semantics in RSL we abstract the templates.

5.1 The Class

We start by formalizing the semantics of a class –the building block of class diagrams. In UML, a class is a description of a set of objects or class instances that share the same structure and behavior. The structure is captured by the attributes, and the behavior by the operations. When considering an instance or object of a class, we can capture its structure and behavior in an RSL module, where we give the corresponding specification. We define an RSL abstract type to specify the class sort. This RSL abstract type denotes the set of all possible class instances or objects. During our first semantic analysis, we considered to specify it as an RSL record, whose destructors correspond to the attributes of the class. However, when we later analysed the semantics of Generalization, we found that this class sort specification it was incompatible with the use of RSL subtypes for specifying the class sort of a subclass. Section 5.2.2 provides the information on the semantics of Generalization relation.

The attributes correspond to the structural part of the class, so with the corresponding observer defined on the class sort the attribute should be obtained. Consequently, we define an observer on the class sort for each attribute. Each one of these observers takes an instance of the class and returns a value belonging to the corresponding attribute type.

The class operations correspond to the behavioral part of the class. Each operation achieves a particular behavior based on the class structure and, possibly, on other parameters. Therefore, they can be specified as RSL functions, which have their domain in the Cartesian product of the class sort and the corresponding operation parameters, and the range corresponds to the operation result type. When an operation in the class diagram does not return a value, it means

that it performs some behavior based on the class structure and the parameters and possibly changes the class structure. Therefore, in this situation, the RSL function will return the class sort.

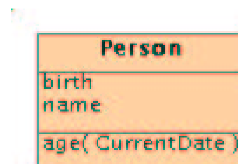

Figure 4: Example of a class in UML

The class "Person" –given in figure 4– has two attributes, one for the name and the other for the date of birth, and one operation that given a date computes the person's age. According to our interpretation, one instance of the class might be specified in RSL as follows:

```
TYPES
object PERSON :
  with TYPES in
  class
    type
          Person
    value
          birth: Person → birth,
          name: Person → name,
          age: CurrentDate × Person → Person
  end
```

In a class, besides the typical operation related to obtain the value of an attribute, it is common to have an operation to modify it, and since, frequently, the update of a given attribute occurs under a given precondition, we decided to generate RSL functions for this purpose. After the RSL specification is generated, their preconditions should be completed by the user. If they are not necessary, then they may be removed. Accordingly, we give the following RSL specification for an instance of "Person":

```
TYPES
object PERSON :
  with TYPES in
  class
    type
          Person
    value
```

birth: Person $\rightarrow$ birth,
name: Person $\rightarrow$ name,

update_birth: birth $\times$ Person $\stackrel{\sim}{\rightarrow}$ Person
update_birth(at, o) **as** o′ **post** birth(o′) = at
  pre preupdate_birth(at, o),

preupdate_birth: birth $\times$ Person $\rightarrow$ **Bool**,

update_name: name $\times$ Person $\stackrel{\sim}{\rightarrow}$ Person
update_name(at, o) **as** o′ **post** name(o′) = at
  pre preupdate_name(at, o),

preupdate_name: name $\times$ Person $\rightarrow$ **Bool**,

age: CurrentDate $\times$ Person $\rightarrow$ Person

end

In UML, it is possible to produce class diagrams at different levels of abstraction. For instance, when we are building a conceptual model, we are concerned with showing some attributes but no information about their types. Therefore, when the type of an attribute or operation parameter is not given in the diagram, we interpret it as an abstract type, and we define a sort in the module TYPES which will hold the specification for all the model types. However –as it was discussed before– when the type of an operation result is not present in the diagram and since class operations act on the object state, we specify the type returned by the RSL function corresponding to the class operation as the class sort.

We have given the semantics for an instance of a class. Now we need to give an interpretation for the class itself. As has been said before, a class defines a set of instances, and if we consider that a class diagram describes all the possible valid states in which a system can be, that is, all the possible valid states of its objects and links (association instances) at a given time, and if we consider a class in the context of a class diagram, we need to define a type that describes the set of all the possible observable states in which the class can be, that is, a set of sets of objects or, in other words, all the possible sets of objects that can be observed at a given moment. In order to describe this fact in RSL, we define a type for specifying all the possible instances or objects of the class. We refer to it as the class container type. A characteristic of the objects is that each object is distinguishable from the others objects of the class, even if they have exactly the same property values. For example, two different instances of a Person can have the same name and age, but they are different as instances, that is they have an identity. Consequently, we define the class container type as a map from a set of object identifiers to the class sort. For example, if we consider the class "Person" again, we define the type Persons that represents all the possible instances of class "Person" as follows:

type

Persons = Person_Id $\overrightarrow{m}$ Person

The specification for a class is given in a new RSL module (in this example "PERSONS"), which uses the module that has the specification for an object (in this example "PERSON"). The whole specification of the class could be contained in only one module, however we have decided to split it in two modules in order to gain in clarity and maintainability.

```
PERSON
object PERSONS :
  with TYPES in
  class
     type
       Person = PERSON.Person,
       Persons = Person_Id →m Person
  end
```

In the context of the system described by a class diagram, for each class in the class diagram, new objects can be created and existing objects can be destroyed or modified. Therefore, some typical functions that operate on the set of instances of each class are defined.

```
PERSON
object PERSONS :
  with TYPES in
  class
    type
      Person = PERSON.Person,
      Persons = Person_Id →m Person

    value
      empty_Person : Persons = [ ],

      add_Person :
        Person_Id × Person × Persons ~→ Persons
      add_Person(id, o, c) ≡ c † [ id ↦ o ]
      pre ∼ isin_Person(id, c),

      del_Person : Person_Id × Persons ~→ Persons
      del_Person(id, c) ≡ c \ {id}
      pre isin_Person(id, c),

      isin_Person : Person_Id × Persons → Bool
```

```
        isin_Person(id, c) ≡ id ∈ dom c,

        get_Person : Person_Id × Persons ~→ Person
        get_Person(id, c) ≡ c(id) pre isin_Person(id, c),

        update_Person :
           Person_Id × Person × Persons ~→ Persons
        update_Person(id, o, c) ≡ c † [id ↦ o]
        pre isin_Person(id, c)
end
```

5.2 Relationships

When we built a class diagram, we identify a number of classes that collaborate and interact among them. This fact might be modeled in UML by using different kinds of relationships. In the next subsections we treat each of them.

5.2.1 Association

The association is the most common relationship among classes. Although it is possible to found associations among any number of classes –usually referred as N-ary associations– the more frequently used is the binary association. In figure 5 below we can see, for instance, a binary association between class “Item” and class “Copy” named “has_copies”. It means that instances of class “Item” might be related with instances of class “Copy” through “has_copies” and vice versa.

The interpretation for an association among n classes is a mathematical relation among the sets of instances corresponding to each class. So, an association describes a set of associations instances, called in UML links. Therefore, an association can be viewed as a set of links, where each link is a n-tuple that represents an instance of n objects related by the association.

There are two possibilities to specify associations in RSL, besides their arities. One possibility consists of specifying the association as a class that has n object-valued attributes. An object-valued attribute is an attribute whose type corresponds to the type used to identify an object among all the instances of a class. The other possibility consists in splitting the relationship among the involved classes, that is, each class contains only one object-valued attribute for recording the n-1 remaining elements of the n-tuple, i.e. the relation is divided into n functions.

Since, in UML, an attribute whose type is a class is viewed as a shorthand for a composite association –a particular kind of association (see section 5.6)–, technically there is no difference between attributes and associations. Therefore, we adopt the second approach to specify all kind

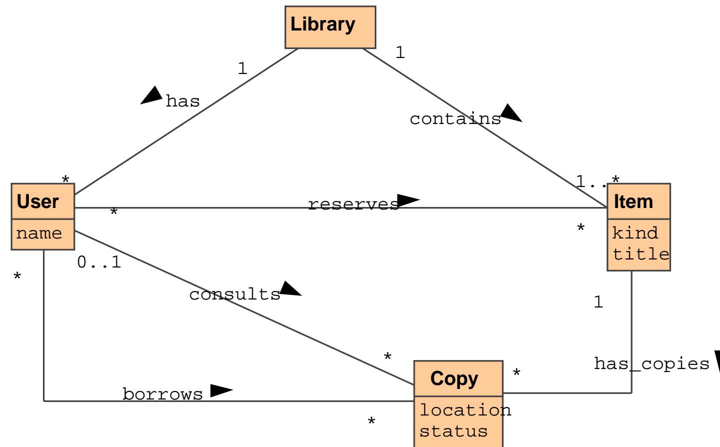

Figure 5: An UML class diagram

of associations as object valued attributes. Note that the first approach presents the following problem: the association is separated from the classes, so if we want to reuse one of the involved classes, we have to know or be reminded that the association is external to them in order to carry this information into the new system. Since this coupling information is not part of the class it is possible that it could be lost.

According to the adopted approach, for the binary association "has_copies" in figure 5, we define two observers: one in class "Item" to retrieve all the existing copies for a given item, and another in class "Copy" to obtain the corresponding item for a given copy.

TYPES
object ITEM :
  with TYPES **in**
  class
    type Item

    value
      title : Item $\rightarrow$ title,
      kind : Item $\rightarrow$ kind,
      ...
      has_copies : Item $\rightarrow$ Copy_Id-**set**,

      update_has_copies : Copy_Id-**set** $\times$ Item $\stackrel{\sim}{\rightarrow}$ Item
      update_has_copies(a, o) **as** o′ **post** has_copies(o′) = a
      pre preupdate_has_copies(a, o),

      preupdate_has_copies : Copy_Id-**set** $\times$ Item $\rightarrow$ **Bool**,

```
        ...
  end


TYPES
object COPY :
  with TYPES in
  class
    type Copy

    value
      status: Copy → status,
      location: Copy → location,
      ...
      has_copies: Copy → Item_Id,

      update_has_copies: Item_Id × Copy ~→ Copy
      update_has_copies(a, o) as o′ post has_copies(o′) = a
      pre preupdate_has_copies(a, o),

      preupdate_has_copies: Item_Id × Copy → Bool,
      ...
  end
```

As we did for the attributes, we define also all the functions for updating the links. Therefore, we also include in the specification of an item and a copy the corresponding values for that.

Note that in the case of the class “Item”, the returned types for has_copies is a set of object identifiers of class “Copy”, while in “Copy” has_copies only returns an object identifier of class “Item”. We make here a structural distinction because the multiplicities at the association ends are different. Association multiplicities are treated separately later in section 5.8.1.

The Association Class

In UML, associations that have attributes and/or relationships are represented by association classes –associations with the properties of a class. For example, figure 6 shows the association class “Job” between “Employee” and “Company”. “Job” could have its own attributes and operations.

Following the semantics expressed in the OMG Unified Modeling Language Specification [22], in general, an association class relating n classes could be decomposed into an association among the n involved classes plus a new class –that represents the association class– which is related to the n classes through new associations, as we show in figure 7.

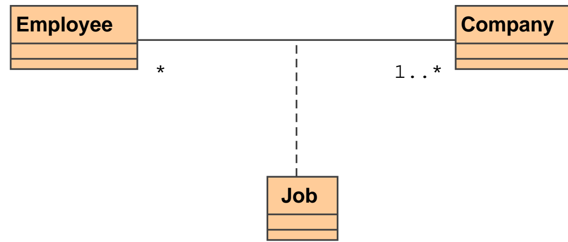

Figure 6: An association class

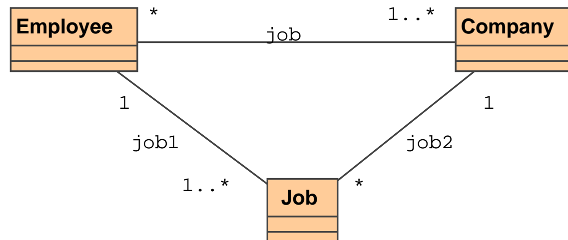

Figure 7: Decomposition of an association class

However, it is necessary to constrain the association "job" and the two new associations "job1" and "job2" to satisfy job = job1 ∘ job2, where ∘ denotes the composition operation of relations.

$$
\begin{array}{l}
(\forall\ \mathrm{id1} : \mathrm{Employee_Id} \bullet \\
\quad (\mathrm{id1} \in \mathbf{dom}\ \mathrm{employees(s)} \Rightarrow \\
\quad\quad (\forall\ \mathrm{id2} : \mathrm{Job_Id} \bullet \\
\quad\quad\quad \mathrm{id2} \in \mathrm{EMPLOYEE.job1(employees(s)(id1))} \Rightarrow \\
\quad\quad\quad\quad \mathrm{JOB.job2(jobs(s)(id2))} \in \\
\quad\quad\quad\quad\quad \mathrm{EMPLOYEE.job(employees(s)(id1))))))} \wedge \\
(\forall\ \mathrm{id1} : \mathrm{Company_Id} \bullet \\
\quad (\mathrm{id1} \in \mathbf{dom}\ \mathrm{companys(s)} \Rightarrow \\
\quad\quad (\forall\ \mathrm{id2} : \mathrm{Job_Id} \bullet \\
\quad\quad\quad \mathrm{id2} \in \mathrm{COMPANY.job2(companys(s)(id1))} \Rightarrow \\
\quad\quad\quad\quad \mathrm{JOB.job1(jobs(s)(id2))} \in \\
\quad\quad\quad\quad\quad \mathrm{COMPANY.job(companys(s)(id1))))))}
\end{array}
$$

The same association class could be modeled by the diagram shown in the figure 8. Other authors express the semantics for association class in this way (see [18]).

However, following [25] (see page 157), this is not enough. To comply with these semantics,

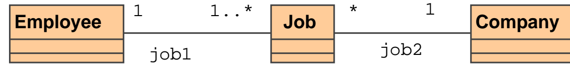

Figure 8: Decomposition of an association class

we have to constrain the associations shown in figure 7 to avoid two different instances of the association class relating the same tuple of objects.

$$
\begin{array}{l}
(\forall \text{ id1, id2 : Job_Id} \bullet \\
\quad (\text{id1} \in \textbf{dom} \text{ jobs(s)} \wedge \\
\quad\quad \text{id2} \in \textbf{dom} \text{ jobs(s)} \wedge \text{id1} \neq \text{id2}) \Rightarrow \\
\quad\quad\quad (\text{JOB.job1(jobs(s)(id1)), JOB.job2(jobs(s)(id1)))} \neq \\
\quad\quad\quad\quad (\text{JOB.job1(jobs(s)(id2)), JOB.job2(jobs(s)(id2))))}
\end{array}
$$

This decomposition process could be also used to deal with associations with arity greater than two. For example, 3-ary association depicted in figure 9 could be transformed into three associations between the three involved classes, as figure 10 shows.

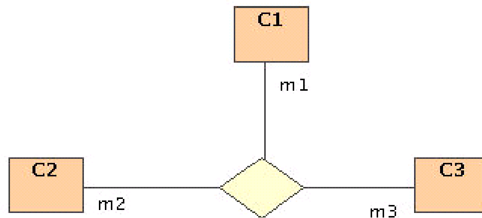

Figure 9: Example of N-ary association

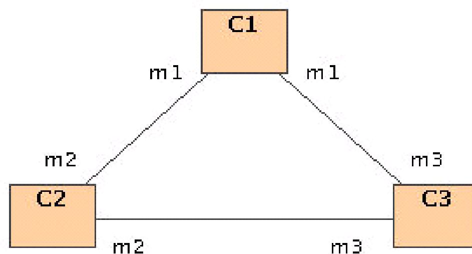

Figure 10: Decomposition of N-ary association

5.2.2 Generalization

In UML it is possible to express inheritance among classes by means of the use of generalization relationships. In UML inheritance, a child class inherits the structure and behavior of the parent. Furthermore, the child may add new structure and behavior or it may even modify the behavior.

Figure 11 below shows an example of the use of generalization between classes "Person" and "User".

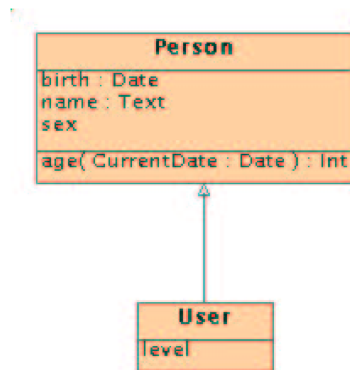

Figure 11: Example of Generalization in UML

"User" inherits all the structure and behavior of "Person" and adds new structure by adding the attribute "Level".

We analysed two different approaches for representing inheritance using RSL. Since RSL does not provides dynamic binding, i.e. the object capability to change its type at run time, only subtype polymorphism could be supported (the possibility of applying a function defined for a type to a value of a subtype of that type). We first considered the possibility of specifying the class sort corresponding to the subclass as a record which holds the structure of the superclass plus the new added structure.

type
  User ::
    person : PERSON.Person ↔ replace_person
    level : Level ↔ replace_level

The type User incorporates all the structure of a Person. However, in order to "inherit" in RSL the superclass behavior it is necessary to define all the superclass methods again in the subclass.

PERSON

```
object USER :
   with TYPES in
   class
      type
         User ::
            person : PERSON.Person ↔ replace_person
            level : Level ↔ replace_level

      value
         name : User → Name
         name(user) ≡ PERSON.name(person(user)),

         birth : User → Birth
         birth(user) ≡ PERSON.birth(person(user)),

         update_name : Name × User → User
         update_name(name, user) ≡
            mk_User(
                PERSON.update_name(name, person(user)), level(user)),

         update_birth : Birth × User → User
         update_birth(birth, user) ≡
            mk_User(
                PERSON.update_birth(birth, person(user)), level(user)),

         age : Date × User → Age
         age(date, user) ≡ PERSON.age(date, person(user)),

         update_level : Level × User ~→ User
         update_level(level, user) ≡
            replace_level(level, user)
         pre preupdate_level(level, user),

         preupdate_level : Level × User → Bool
      end
```

UML also allows multiple inheritance among classes. In multiple inheritance the subclass inherits all the properties of the superclasses, i.e., all the attributes and operations defined for the superclasses. The subclass can even add new structure and behavior or redefine the behavior.

The same approach explained above for single inheritance could be applied when there is multiple inheritance, that is, in the subclass we define a record in RSL where we incorporate the structure of the superclasses plus all the subclass' attributes, and we generate all the superclass' methods plus the subclass' methods as well. When in multiple inheritance there are two superclasses with

common attribute names and the same maximal type, in the subclass we get only one attribute whose type is the minimal type among the types of the superclasses. When the maximal types are different we have the typical problem of multiple inheritance: clash of names. The same problems can arise from overlapped behavior.

However, this approach presents several problems. This is because there is no way of guaranteeing that, in the future development, when the developer will work on the RSL specification, some inconsistencies between the superclass and the subclass can occur. For instance, new behavior in the superclass could be added and omitted in the subclass, making undefined each reference to the new method when applied to an object of the subclass. Besides, since the class sorts defined for each class in the class hierarchy are disjoint, each class has its own class container and consequently the constraints for a superclass must be repeated for each subclass in the hierarchy, leading to possible inconsistencies again.

To solve this problem we instead use subtypes: the type that denotes the set of all the instances of the subclass is defined as an RSL subtype of the class sort that corresponds to the superclass. Therefore, all the functions that operate on the type defined for the superclass are available for the subtype. New attributes and operations may be added to the subclass by defining the corresponding functions on the subclass type and the superclass method can be redefined in the subclass by redefining the corresponding function names. This is possible because each class has its own module, so no problem of function redefinition occurs in RSL.

```
PERSON
object USER :
  with TYPES in
  class
    type
      Person = PERSON.Person,
      User = {| o : Person • is_a_User(o) |}

    value
      level : User → level,

      update_level : level × User ~→ User
      update_level(at, o) as o′ post level(o′) = at
      pre preupdate_level(at, o),

      preupdate_level : level × User → Bool,
      is_a_User : Person → Bool
  end
```

The type corresponding to the type container of the subclass is defined as a subtype as well. In this case, as a subtype of the type used to specify the class container of the superclass.

```
type
  Users =
    {| super : PERSONS.Persons •
       (∀ id : Person_Id •
          id ∈ dom super ⇒ USER.is_a_User(super(id))) |}
```

The functions to operate on the class container are defined as usual.

```
USER, PERSONS
object USERS :
  with TYPES in
  class
    type
      User = USER.User,
      Users =
        {| super : PERSONS.Persons •
           (∀ id : Person_Id •
              id ∈ dom super ⇒ USER.is_a_User(super(id))) |}

    value
      empty_User : Users = [ ],

      add_User : User_Id × User × Users ~→ Users
      add_User(id, o, c) ≡ c † [id ↦ o]
      pre ~ isin_User(id, c),

      del_User : User_Id × Users ~→ Users
      del_User(id, c) ≡ c \ {id}
      pre isin_User(id, c),

      isin_User : User_Id × Users → Bool
      isin_User(id, c) ≡ id ∈ dom c,

      get_User : User_Id × Users ~→ User
      get_User(id, c) ≡ c(id) pre isin_User(id, c),

      update_User : User_Id × User × Users ~→ Users
      update_User(id, o, c) ≡ c † [id ↦ o]
      pre isin_User(id, c),

      consistent : Users → Bool
      consistent(c) ≡
        (∀ id : User_Id •
```

id ∈ **dom** c ⇒ USER.consistent(c(id)))

end

Figure 12 below shows the dependency graph among the modules that have the specification for the superclass "Person" and the subclass "User".

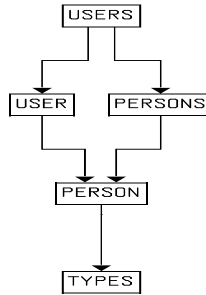

Figure 12: RSL module dependency graph

Since we interpret the class sort of a subclass as an RSL subtype of the superclass sort, it is not possible to represent multiple inheritance. However, in general, multiple inheritance must be used carefully because of problems that could arise from multiple parents with overlapped behavior or structure, besides of the loss of maintainability: the more superclasses, the harder it is to know what comes from where and how the changes affect the lattice. Multiple inheritance may be replaced by delegation, where the subclass inherits only from one of the candidate superclasses and incorporates into its structure the other superclasses to obtain their structure and behavior. In figures 13 and 14 an example of multiple inheritance and the transformation into a delegation are shown.

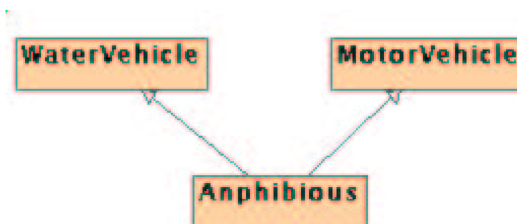

Figure 13: Multiple inheritance

Root Classes

UML allows one to constrain a class with the property {root}. This means that such a class may not have a superclass. It does not seem to be very useful, except when there is a multiple inheritance lattice and one wants to designate the head of each hierarchy.

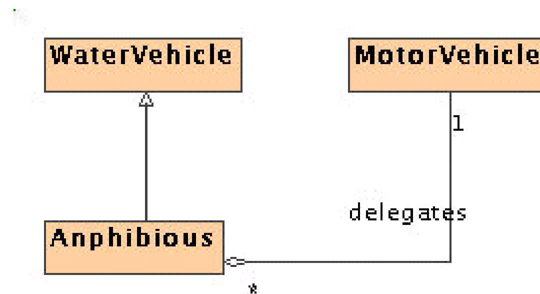

Figure 14: Delegation

This is a constraint captured in the syntax, so there is no reason to give its dynamic semantics.

Leaf Classes

Property {leaf} is used in UML to point out that a given class does not have children. When we give the semantics of a class having this property attached, we can fix the final structure of the sort class because no inheritance is possible from this class, i.e. the type corresponding to the class it is no more specified as a sort but as a more concrete RSL type: a record. For instance, if the class "Person" in figure 4 on page 25 is a leaf class, we specify it in RSL as follows:

TYPES
object PERSON :
  with TYPES **in**
  class
    type
      Person ::
        birth : birth $\leftrightarrow$ replace_birth
        name : name $\leftrightarrow$ replace_name

    value
      update_birth : Date $\times$ Person $\xrightarrow{\sim}$ Person
      update_birth(at, o) $\equiv$ replace_birth(at, o)
        pre preupdate_birth(at, o),
      ...
      age : CurrentDate $\times$ Person $\rightarrow$ Person
  end

5.2.3 Dependency

Dependency is a binary relationship. When used between two classes it expresses that one class uses the other in some way. It is possible to use a dependency between two classes when, for example, there exists the knowledge that one class is going to use the other as the parameter type in one of its operations, even if the operation has not yet been given in the class, or for expressing, for example, that the class is used locally in a operation. So, this kind of relationship can have a variety of meanings that can not be directly derived from a class diagram.

5.3 Class Diagrams

A class diagram is useful for building static models of a system expressed in terms of classes and relationships among them. When we use class diagrams for building conceptual models, they reflect the concepts and relationships that are present in the application domain, while when they are used at design level, they reflect a solution-oriented structure.

A class diagram represents the set of all the possible Object Diagrams. An object diagram models instances of the elements contained in the corresponding class diagram, i.e. objects and links. It is like a snapshot and shows one of the possible configurations for a given class diagram. So, an object diagram is essentially an instance of a class diagram.

We say that a class diagram describes the set of all its object diagrams or the state space of the system, and that an object diagram corresponds to one system state. Therefore, we define a type in RSL to specify the state space of the system as a record whose fields are all the present classes in the class diagram that can have instances associated to them. For example, for the class diagram given in figure 5 on page 29 we have:

```
ITEMS, USERS, LIBRARYS, COPYS
object S :
  with TYPES in
  class
    type
      Sys ::
        items : ITEMS.Items ↔ replace_Items
        users : USERS.Users ↔ replace_Users
        librarys : LIBRARYS.Librarys ↔ replace_Librarys
        copys : COPYS.Copys ↔ replace_Copys

    value
      update_Items : ITEMS.Items × Sys ~→ Sys
      update_Items(c, s) ≡ replace_Items(c, s)
      pre preupdate_Items(c, s),
```

```
      preupdate_Items : ITEMS.Items × Sys → Bool,
        ...

      update_Copys : COPYS.Copys × Sys ~→ Sys
      update_Copys(c, s) ≡ replace_Copys(c, s)
      pre preupdate_Copys(c, s),

      preupdate_Copys : COPYS.Copys × Sys → Bool
  end
```

In this example, all the classes have the corresponding field in the record Sys to hold their class containers, but note that this is not always the case: since a subclass is a specialization of a superclass, it shares the class container with its superclass which, in turn, if it is a subclass does the same with its corresponding superclass, and so on. A class having several subclasses shares the container with all of them. That is, the class container of the subclasses are subsets of the superclass container. In this way, we guarantee that each desirable established property for the superclass, expressed by means of constraints on the class container, is inherited by the subclasses. For example, the type "Sys" defined for the class diagram shown in figure 11 on page 33 has only the class container for the superclass "Person". The class container for "User" is obtained from the result of applying a function on the superclass container:

```
USERS, PERSONS
object S :
  with TYPES in
  class
    type
      Sys :: persons : PERSONS.Persons ↔ replace_Persons

    value
      users : Sys → USERS.Users
      users(s) ≡
        persons(s) /
        {id |
         id : User_Id •
           id ∈ dom persons(s) ∧
           USER.is_a_User(persons(s)(id))},
     ...
end
```

The type "Sys" describes all the possible instances of the system being modeled by the class diagram. Since we are interested only in consistent systems, it is necessary to express consistency.

This is achieved by giving a collection of axioms to express the property that all top-level state-changing functions maintain the system in a consistent state. We capture all the model invariants in a boolean function named "consistent" which is defined as the conjunction among all the necessary predicates to express all the model constraints . So, all the consistency constraints that can be directly derived from the diagram are expressed in this function. OCL constraints [27] –given by the user explicitly in the UML class diagram– could also be expressed as invariants in RSL through a translation mechanism. Among those derived from the class diagram we have multiplicity constraints, identifiers of objects stored as part of the attributes of a class, and bidirectional navigations. Other kind of consistency constraints, such as for example more than one access path to the same information or potentially circular relations cannot be automatically derived from the diagram because they depend on the semantics of the relationship. This last kind of constraints should be expressed explicitly by the user. Class diagram consistency is treated in more detail in section 5.8.

Having given the semantics for a class diagram and the associations, an important issue must be pointed out:
RSL is a structured language, while object-oriented designs might not be. In a class diagram, the dependencies between the classes may be circular, and in RSL the module structure must be hierarchical. When we built design class diagrams, one important element is the methods. A method not only allows one to operate on the object attributes but also makes possible the interaction with other objects. In our RSL specification for a class diagram, the top level module is the only one that has the possibility of performing this kind of responsability. Therefore all the function signatures corresponding to the methods – besides of being given inside the corresponding module– must be given in the top level module as well. Each one of these top level functions –when it just operates on the attributes of the class– should be defined by the only reference to that function defined inside the class module. However, when it depends in some way on other classes, it has to be defined in the top level module.

5.4 Initial value of an attribute

There exists in UML the possibility of assigning an initial value to a class attribute. This initial value is assigned to the attribute each time that an instance of the class is created. Initial values in UML are optional, and since they can even be modified by an explicit constructor it is not mandatory to check –each time a new object is created– that its attributes have been assigned with their corresponding initial values, since sometimes this is not the intention. Therefore, when initial values are given in a UML class diagram they are ignored in RSL.

5.5 Association Navigation

Associations between two classes, when no navigation adornment is given, are assumed to be bi-directional, i.e. they can be used to navigate from objects of one class to objects of the other class and vice versa. However, there are situations in which the designer wants to restrict

navigation to be only in one direction. This can be expressed in UML by using a navigation adornment at the corresponding association end.

In our interpretation of class diagrams, when a navigation adornment is present in only one of the association ends of a binary association, instead of defining one function in each class to retrieve the related objects –as in all the previous examples– only one is defined.

UML is not clear about the meaning of navigation in N-ary associations. For example the diagram given in figure 15 shows a general 3-ary association with a navigation adornment at the end corresponding to class C2.

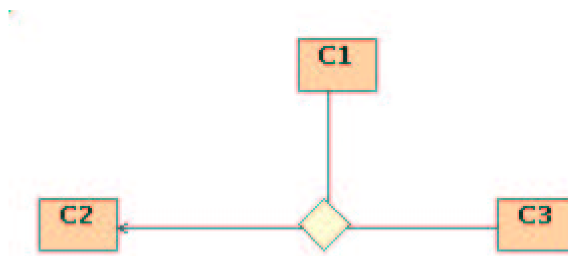

Figure 15: Association Navigation in N-ary associations

One possible interpretation might be that all the ends are departure and arrive ends except those having an arrow, which are considered as only-arrive ends. Under such an interpretation, in the example, it is possible to navigate from objects in C1 to objects in C3 and C2, and also from objects in C3 to objects in C1 and C2, but it is not possible to navigate from C2 to anywhere.

Another interpretation is that only the ends that have a navigation adornment are arrive and departure ends, while the others are considered as only-departure. According to the last interpretation, it is only possible to navigate from objects in C1 and from objects in C3 to objects in C2.

If we add an arrow at the end corresponding to class C3, according to the first interpretation it is only possible to navigate from C1 to C2 and C3 while in the second interpretation it is possible to navigate from C1 to C3 and C2, from C2 to C3 and from C3 to C2.

We have adopted the second one, because when the adornment is present in all the ends, according to the first interpretation it is impossible to navigate at all, while in the second one it is possible to navigate from any end to any other.

The adopted interpretation works for relations of any arity. For example, in the case of binary associations, we have : when the arrows are present in both ends, both are considered arrive an departure ends, therefore it is possible to navigate in both directions. When only the navigabilty is in one direction, i.e. only one arrow is present, since the end without arrow is considered only-departure and the other arrive and departure, it is possible to navigate from the first one to the

second one, but it is not possible do it in the opposite direction because of –although the second one is a departure end– the first one is only-departure (and not arrive).

5.6 Composition and Aggregation

Other association end adornments that is possible to use in UML are composition and aggregation adornments. Both are used for modeling a "whole/part" relationship, in which one class represents the "whole" which consists of smaller things (the "parts").

As [5] establishes, aggregation is purely conceptual, and does no more than distinguish the whole from the parts. It does not change the meaning of association. Therefore, in our interpretation aggregation between a component class and an aggregate class is equivalent to a general association among the involved classes.

On the other hand, composition has a more strong meaning. Composition is a form of aggregation with coincident lifetime of the parts with the whole, i.e. the parts may be created after the whole, and can also be explicitly removed before the whole. However, if the whole is destroyed they die with it. Because of that, the multiplicity of a composite end –unlike an aggregate end– must be always at most one (it cannot be shared by different instances of the owner class). UML does not allow aggregation and composition in associations with arity greater than two.
In figure 16 an example of a composition between the classes "Company" and "Department" and a recursive composition on "Department" are shown.

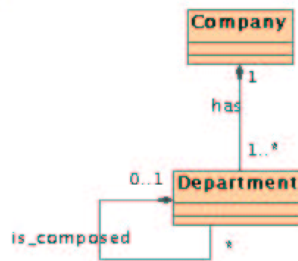

Figure 16: Composition

Structurally, a composition is equivalent to a general association which has at least one end with multiplicity at most equal to one. Therefore, the structure here is not a distinctive characteristic. However, when we formalize the dynamic aspects of a composition, we make a distinction between a general association and a composition based on the property of coincident lifetimes of the whole and the parts. We express this property by means of a postcondition in the remove function of the whole. It assures that always when a whole is deleted all the parts that are currently associated are also deleted.
Given the composition "has" between the part "Department" and the whole "Company" shown in figure 16, we specify the coincident lifetime property as follows:

```
del_Department : Department_Id × Sys ~→ Sys
del_Department(id, s) as s″ post
   (∃
      s′ : Sys, new_whole : DEPARTMENTS.Departments,
      new_parts : DEPARTMENTS.Departments,
      parts : Department_Id-set
   •
      parts =
         DEPARTMENT.is_composed1(departments(s)(id)) ∧
      new_parts = departments(s) \ parts ∧
      s′ = update_Departments(new_parts, s) ∧
      new_whole =
         DEPARTMENTS.del_Department(id, departments(s′)) ∧
      s″ = update_Departments(new_whole, s))
pre can_del_Department(id, s),

can_del_Department : Department_Id × Sys → Bool
can_del_Department(id, s) ≡
   DEPARTMENTS.isin_Department(id, departments(s)) ∧
   (∃
      s′ : Sys, new_whole : DEPARTMENTS.Departments,
      new_parts : DEPARTMENTS.Departments,
      parts : Department_Id-set
   •
      parts =
         DEPARTMENT.is_composed1(departments(s)(id)) ∧
      new_parts = departments(s) \ parts ∧
      preupdate_Departments(new_parts, s) ∧
      s′ = update_Departments(new_parts, s) ∧
      new_whole =
         DEPARTMENTS.del_Department(id, departments(s′)) ∧
      preupdate_Departments(new_whole, s))
```

In UML, when the type of an attribute corresponds to a class sort, this attribute is, in effect, a composition relationship between the class and the class of the attribute, or, in other words, the attribute is a shorthand for composition. Consequently, the semantics in RSL for this kind of attribute is given in the same way that for a composition relationship which is navigable only from the "whole" to the parts.

5.7 Parameterized Classes

UML allows the use of templates or parameterized classes, i.e. classes that describe a family of classes. A template class cannot be used directly, but it must be instantiated first, so template

classes do not contain objects. Attributes and operations within a template may be defined in terms of the formal parameters so they become bound when the template itself is bound to actual values.

Since a template instance is fully specified by its template, its content may not be extended (declaration of new attributes or operations are not permitted); consequently it only may be the target of an association or be a superclass. Since template classes cannot have instances, it can only have one direction associations from itself to another class. They can also be a subclass of another class, but they cannot be a superclass, only its instances can. An example of a template class and two different instantiations are shown in figure 17.

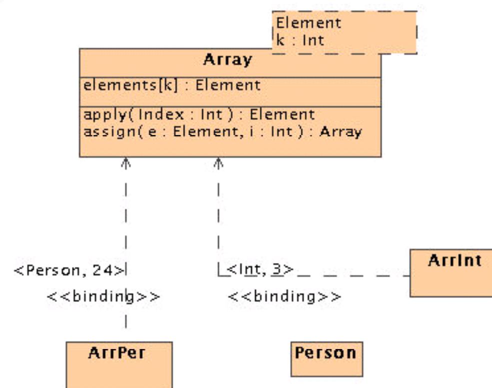

Figure 17: Template and Instantited Classes

The figure shows a template class for one array of k elements and the instantiation of one array of 3 integers and another of 24 persons.

When in a parameter expression, only the name is given (as Element), it is assumed to be a type expression that resolves to a valid data type (as Person or **Int**). Otherwise, it must resolve to a valid value expression (like 3 or 24 for **Int**).

A template class may be specified in RSL in the same way as a concrete class but using a parameterized scheme whose parameters corresponds to the class's parameters.

```
TYPES
scheme ARRAY_(
    FPAR :
        with TYPES in
        class
            type Element

            value
```

```
        k : Int
      end) =
   with TYPES in
   class
      type
         Array,
         Elements = FPAR.Element-set

      value
         elements : Array → Elements,

         update_elements : Elements × Array ~→ Array
         update_elements(at, o) as o′ post elements(o′) = at
         pre preupdate_elements(at, o),

         preupdate_elements : Elements × Array → Bool,

         apply : Int × Array → FPAR.Element,

         assign : FPAR.Element × Int × Array → Array_Id,

         consistent : Array → Bool
         consistent(o) ≡ card elements(o) = FPAR.k
   end
```

The parameter "k" is used for expressing a property on the multiplicity of the attribute. This will be more clear in section 5.8 where consistency checks are treated separately.

Since a template class cannot be used directly but only through its instantiations, the specification for its class container is not defined. However, this is done for each instantiated class that is present in the class diagram since they may have direct instances associated to them and, consequently, they can change the system state.

The semantics in RSL for the instantiation of a parameterized class is given by the instantiation of the corresponding parameterized scheme with its corresponding types and values. So, to instantiate the array of 3 integers we instantiate the scheme ARRAY_ with Element equal to Int and k equal to 3 as follows:

```
ARRAY_
scheme ARRINT_ =
   with TYPES in
   extend
      class
```

```
      object
         APAR_ArrInt :
            class
               type Element = Int

               value
                  k : Int = 3
            end
   end
with
extend ARRAY_(APAR_ArrInt) with class type ArrInt = Array end
```

and –as usual– we create an RSL object to represent the model corresponding to the specification of one object of class "ArrInt".

```
ARRINT_
object ARRINT : ARRINT_
```

To instantiate the array of Persons and assuming that "Person" is a class, Element is replaced by the object identifier of class "Person".

```
ARRAY_
scheme ARRPER_ =
   with TYPES in
   extend
      class
         object
            APAR_ArrPer :
               class
                  type Element = Person_Id

                  value
                     k : Int = 24
               end
      end
   with
   extend ARRAY_(APAR_ArrPer) with class type ArrPer = Array end
```

If class "Person" does not exist, then an abstract type Person is defined in TYPES, and we would have instead the type Element defined as the abstract type Person:

```
ARRAY_
scheme ARRPER_ =
  ...
     type Element = Person
  ...
```

Like concrete classes, instantiated classes can have objects, therefore the module that holds the specification for the class container is specified as usual, for each instantiated class. For example, the class container for the class "ArrInt" is given by the following RSL code:

```
ARRINT
object ARRINTS :
  with TYPES in
  class
    type
      ArrInt = ARRINT.ArrInt,
      ArrInts = ArrInt_Id →m ArrInt

    value
      empty_ArrInt : ArrInts = [ ]
        ...

  end
```

When the template class is a subclass, their corresponding instantiated classes share the container with the superclass, otherwise they have their corresponding containers. For example, according to the new class diagram shown in figure 18 the classes "ArrInt" and "ArrPer" share their class container with the class "DataStructure". We only define one function to obtain all the objects corresponding to the subclass by selecting from the set of objects in the superclass container those that belongs to the subclass.

```
PERSONS, DATASTRUCTURES, ARRINTS, ARRPERS
object S :
  with TYPES in
  class
    type
      Sys ::
        persons : PERSONS.Persons ↔ replace_Persons
        datastructures :
          DATASTRUCTURES.DataStructures ↔
          replace_DataStructures
```

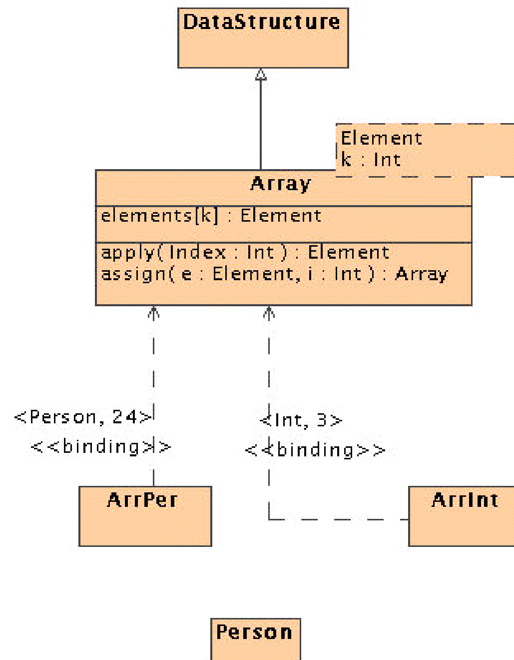

Figure 18: Inheritance plus Template Classes

value

...

arrints : Sys → ARRINTS.ArrInts
arrints(s) ≡
datastructures(s) /
{id |
id : ArrInt_Id •
id ∈ **dom** datastructures(s) ∧
ARRINT.is_an_ArrInt(datastructures(s)(id))},

...

arrpers : Sys → ARRPERS.ArrPers
arrpers(s) ≡
datastructures(s) /
{id |
id : ArrPer_Id •
id ∈ **dom** datastructures(s) ∧
ARRPER.is_an_ArrPer(datastructures(s)(id))},

```
    ...

    consistent : Sys → Bool
    consistent(s) ≡
        PERSONS.consistent(persons(s)) ∧
        DATASTRUCTURES.consistent(datastructures(s)) ∧
        ARRINTS.consistent(arrints(s)) ∧
        ARRPERS.consistent(arrpers(s))

    ...
end
```

5.8 Constraints

Class diagrams as well as their composing elements may have different constraints associated with them. As we want to check consistency on the whole system, that is, to check that all the constraints hold, we define a series of axioms on the top level module in order to check that the system is in a consistent state before and after any state change occurs. For this reason, we define a series of boolean functions in order to express there each one of the different constraints of the class diagram. Inside each module that has been defined either to specify an object or a class, we define one of these boolean functions –that we name "consistent". They allow to check the consistency of one object and the consistency of all the instances of the class, respectively. The last ones make use of the lower level ones, that is those defined for one instance of the class. Furthermore, we define one function "consistent" in the top level module in order to check the consistency of the whole class diagram. This function uses –in turn– all the lower level functions "consistent" defined for each one of the classes, guaranteeing in this way the consistency of the whole system.

The top level axioms use the top level function "consistent" in combination with all those top-level functions that change the system state. We define one axiom for each top level changing system state function.

In which one of the several defined functions "consistent" we put the predicate for expressing a given property depends on what kind of restriction are we checking and on what we want to check it.

In the next sections we present the different constraints derived from a class diagram and we give the corresponding predicates in RSL as well as their locations in the whole specification. In the appendices A to E the different templates that have been abstracted from the previous analysis are given. All the templates corresponding to the constraints that are treated in this section are presented in appendix F.

5.8.1 Multiplicities

When a UML class diagram is built, it is possible to assign multiplicities to its classes, attributes and/or to association ends. A multiplicity is a specification of the range of allowable cardinalities an entity may assume.

A class multiplicity constrains the number of instances that a class may hold. In absence of an indicator the class may hold any number of instances, that is, its multiplicity is "*" whose meaning is precisely "any number".

A multiplicity indicator in an attribute specifies the cardinality of the set of values that the attribute may hold. In the absence of a multiplicity indicator an attribute holds exactly one value.

When a multiplicity indicator is present in an association end, it constrains the number of instances associated to the class in the other association end to satisfy the multiplicity. In absence of a multiplicity indicator at the end, a multiplicity equal to one is assumed.

In figure 19 an example of the different kind of multiplicities is shown. The class "Player" has multiplicity equals to '*' and the class "Team" has a multiplicity of "2..*". They are shown in the top right corner of the class.

Figure 19: Multiplicities

In the example, class "Team" is constrained to have always at least 2 instances, while class "Player" has no restriction, therefore no constraint is applied to its number of instances.

Since a constraint on the class multiplicity refers to the number of instances the class can hold, it is convenient to specify this property in the function "consistent" that is defined in the RSL module that holds the specification for the class. This predicate will be, possibly, in conjunction with others constraints on the class if it corresponds.

According to the way in which we have specified a class, the number of instances of a class is given by the domain cardinality of the map that holds all the instances of the class. Consequently, we express the multiplicity constraint for the class "Team" as follows:

consistent : Teams → **Bool**
consistent(c) ≡

... $\wedge$
card dom c $\geq$ 2

Since the class "Player" has assigned multiplicity "*", no predicate is generated.

The interpretation for the multiplicities at the association ends is expressed in RSL by means of predicates on the cardinality of the set of instances that are reached when we navigate from one association end to another. Since associations are defined as functions in the module that holds the specification of one object of the class, the predicate used for expressing this constraint is placed in the function "consistent" of that module. So, for the association "plays" given in the previous example, we could generate one predicates for "Player" and another for "Team".

consistent: Player $\rightarrow$ **Bool**
consistent(o) $\equiv$ ... $\wedge$ **card** plays(o) = 1

consistent : Team $\rightarrow$ **Bool**
consistent(o) $\equiv$ ... $\wedge$
  card plays(o) $\geq$ 11 $\wedge$ **card** plays(o) $\leq$ 22

However, in order to avoid some predicates, it is possible sometimes to constrain the association multiplicity by construction. Therefore, the association ends are not always sets of objects identifiers, but we take into account their multiplicities. That is, given two classes C1 and C2 related by an association A, when the association's end multiplicity at, for example, C2 is 1 we simply use a type identifier of class C2 (C2_Id) as the type of association A instead of using both a set of identifiers (C2_Id-**set**) and a predicate to check that the set cardinality is 1. If the multiplicity is "0..1" we specify the association type as an RSL variant type without generating a predicate for checking the set cardinality either. In this way, only multiplicities specified as sets generate consistency predicates in the function "consistent" –except the unrestricted "*".

The same approach has been applied for attributes. For example, the attribute "name" in the class "Player" has a multiplicity of 1, so no multiplicity constraint is generated in this case, but "address" that has a multiplicity equal to "0..1", it is specified using a type that allows optional values. Optional values in RSL can be specified using RSL variant types.

type
  Optional_address ==
    no_address | an_address(id : address),
  address

And in the module "PLAYER" –among other definitions– we will have the definition of the function "address" to get the attribute value. The predicate for checking the multiplicity is not generated since it is given by construction.

type Player
  ...
value
  address : Player → TYPES.Optional_Address
  ...

On the other hand, the attribute "t_shirt" of the class "Team" has a multiplicity greater than 1. Because of that, the function used to obtain the value of this attribute is specified as a function that returns a set.

type Team
  ...
value
  t_shirt : Team → Color-**set**
  ...

Therefore, when we add the multiplicity constraint in the function "consistent" defined for the class "Team" we have:

consistent : Team → **Bool**
consistent(o) ≡
  card plays(o) ≥ 11 ∧ **card** plays(o) ≤ 22 ∧
  card t_shirt(o) ≥ 1

In appendix F we present the RSL-templates corresponding to predicates generated for the classes and attributes multiplicities. Also the templates for the association end multiplicities are given.

5.8.2 Attribute and Operation Scope

Attributes in UML may have as scope either the instance or the class. When the scope of an attribute is its class, it means that all the instances of the class must share always the same value. On the contrary, if the specified scope for an attribute is given by the property {instance}

it means that each instance in the class holds its own value for the attribute. The default is instance-scope.

Like attributes, operations may have a scope too. Operations whose scope is the class are known as class operations, examples of them are the class constructors and destructors. As with attribute scopes, here the default is the instance-scope.

For example, let us suppose we want to model in UML a class "Window". An instance of a window has a size and might be visible or not at a given time, while all the windows have a default size and a maximal size. The last two attributes have a class scope while the first ones have an instance scope. If we want to have an operation "number_of_windows" to know how many windows have been created, it is clear that an instance of the class cannot answer to this message. Only the class has this possibility. So, "number_of_windows" should be a class-scope operation. In figure 20 the class "Window" is shown. Class-scope attributes are shown by underlining the name and type expression string. Class-scope operations are shown by underlining the operation.

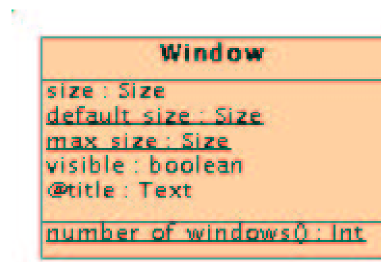

Figure 20: Scopes for attributes and operations

When an attribute has a class-scope, in RSL we have to check that all the instances of the class hold the same value, i.e. for any pair of instances of the class their attribute values must be equal. So, we place the corresponding predicate in the function "consistent" of the module that holds the specification for the class owning the class-scoped attribute.

```
consistent : Windows → Bool
consistent(c) ≡
  ... ∧
  (∀ id1, id2 : Window_Id •
     (id1 ∈ dom c ∧ id2 ∈ dom c) ⇒
        WINDOW.default_size(c(id1)) =
           WINDOW.default_size(c(id2))) ∧
  (∀ id1, id2 : Window_Id •
     (id1 ∈ dom c ∧ id2 ∈ dom c) ⇒
        WINDOW.max_size(c(id1)) =
           WINDOW.max_size(c(id2)))
```

When the scope of an operation is the class, since the operation acts on the class container, we do not generate the corresponding function on an instance of the class, but on the type defined for the class container. Therefore, the value used to specify the operation is placed also in the module that holds the specification of the class. In the example, "number_of_windows" is placed in the module "WINDOWS" as well.

```
WINDOW
object WINDOWS :
   with TYPES in
   class
      type
         Window = WINDOW.Window,
         Windows = Window_Id ⇸ Window

      value
         empty_Window : Windows = [ ],

         add_Window :
            Window_Id × Window × Windows ⥲ Windows
         add_Window(id, o, c) ≡ c † [ id ↦ o ]
         pre ∼ isin_Window(id, c),
            ...

         number_of_windows : Windows → Int,

         consistent : Windows → Bool
         consistent(c) ≡
            (∀ id : Window_Id •
                id ∈ dom c ⇒ WINDOW.consistent(c(id))) ∧
            (∀ id1, id2 : Window_Id •
                (id1 ∈ dom c ∧ id2 ∈ dom c) ⇒
                   WINDOW.default_size(c(id1)) =
                      WINDOW.default_size(c(id2))) ∧
            (∀ id1, id2 : Window_Id •
                (id1 ∈ dom c ∧ id2 ∈ dom c) ⇒
                   WINDOW.max_size(c(id1)) =
                      WINDOW.max_size(c(id2)))
   end
```

5.8.3 Attribute and Association End Properties

Attributes and association ends in UML can have an associated property. The default is {changeable}, i.e. there are no restrictions for modifying the attribute (or the association end) value, unless properties {addonly} or {frozen} are specified. {addonly} only can be used with attributes and association ends with multiplicity greater than one. Its meaning is that additional values may be added, but once created, a value may not be removed or changed. The property {frozen} is used with those attributes whose value may not be changed once an instance of the object that owns the attribute has been created. The same happens with the association ends that has the property {frozen}.

Our default semantics in RSL for attributes and association ends allows the update at any time of attributes and association ends. Therefore, when an {addonly} or {frozen} property is associated to an attribute (or association end), we must check that the attribute (or association end) has only added new values or it has not changed at all, respectively. Since this checking involves the system state before and after a function changes it, this property must be checked at that points. Consequently, the axioms for checking consistency before and after a top-level system state changing function takes places should also check that the property holds for all the instances of the class.

For example, let us suppose that we have the axiom below for checking that the top-level function update_windows –used for changing the "Window" class container– preserves consistency.

axiom
$\forall$ s : Sys, c : WINDOWS.Windows •
  update_Windows(c, s) **as** s′ **post**
    consistent(s′)
  pre consistent(s) $\wedge$ preupdate_Windows(c, s)

If we want also to check the property {frozen} for the window attribute "title" of the class "Window" given in figure 20 (page 54), we need to add a predicate for checking that the value of attribute "title" for each object of class "Window" remains unchanged after the system state has changed. Therefore, the axiom above becomes:

axiom
$\forall$ s : Sys, c : WINDOWS.Windows •
  update_Windows(c, s) **as** s′ **post**
    consistent(s′) $\wedge$ frozenAtts_in_Window(s′, s)
  pre consistent(s) $\wedge$ preupdate_Windows(c, s)

Where "frozenAtts_in_Window" is defined as a boolean function on the previous system state s and the later s′.

value
frozenAtts_in_Window : Sys × Sys → **Bool**
frozenAtts_in_Window(s′, s) ≡
(∀ id : Window_Id •
(id ∈ **dom** windows(s) ∧
id ∈ **dom** windows(s′)) ⇒
WINDOW.title(windows(s)(id)) =
WINDOW.title(windows(s′)(id)))

Since a frozen attribute may not be changed, the corresponding function for updating the attribute, in this example "update_title", is not generated.

Note that a frozen attribute is not equivalent to a constant. Constants in UML may be modeled by means of attributes which not only have associated the property {frozen} but furthermore have the class as scope.

5.8.4 Abstract Classes and Abstract Operations

In UML it is possible to specify in a class hierarchy that a class is abstract, i.e., it is a class that cannot have direct instances associated with it because its behavior is not completely defined. It can only be used as a base for other subclasses. Abstract classes in UML are shown with their names in italics.

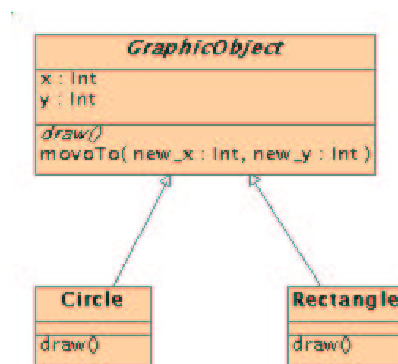

Figure 21: Example of abstract class

Our interpretation in RSL is the same as for a concrete class, the only difference is that –since objects will never be created– we do not need to specify all the functions that operates on the class container. However, we need to express a constraint on the system s in order to assure that no instances of the abstract class have been created. This means that all the instances that can be in the class container corresponds to one of the concrete subclasses of the abstract class.

This property can be expressed as the union of the abstract subclass containers is equal to the abstract class container.

```
consistent : Sys → Bool
consistent(s) ≡ ... ∧
   dom graphicobjects(s) =
      dom circles(s) ∪ dom rectangles(s)
```

Abstract class are classes that have at least one abstract operation. It means that the operation is incomplete and cannot be used, therefore an implementation must be given by a subclass. In RSL, the semantics of an abstract operation is given by hiding the operation name outside of the module. So, outside of the class module only references to the implementations given in the subclasses can occur. In the previous example, operation "draw" is abstract. To avoid the use of "draw" we hide its name.

```
TYPES

object GRAPHICOBJECT :
   with TYPES in
   hide draw in
   class
      type GraphicObject

      value
         x : GraphicObject → Int,
         y : GraphicObject → Int,
            ...
         draw : GraphicObject → GraphicObject,
            ...
   end
```

5.8.5 Existence and Bi-navigation constraints

In section 5.5, we discussed the way in which the association navigation is treated. As we have seen, each time that it is possible to navigate from a class C1 to another class C2 through a given association an observer for the association is generated in the specification given for an object of the class C1. However, since these observers return object identifier(s), a predicate for checking that such object identifiers actually exist in the class container is needed.

For example, if we consider the association "has_copies" between the classes "Copy" and "Item" in figure 5 on page 29, we want to be sure that given a copy, the item identifier returned by

the function "has_copies" –defined in the module that holds the specification of an instance of "Copy"– actually exists in the container of the class "Item". Since this checking involves references to the containers of both classes, it must be placed in a module that has access to both containers, that is the top level module. Therefore, in the function "consistent" of the top level module we add the following predicate:

$$
\begin{array}{l}
(\forall\ \text{id1} : \text{User_Id},\ \text{id2} : \text{Copy_Id} \bullet \\
\quad (\text{id1} \in \textbf{dom}\ \text{users(s)} \wedge \\
\quad\quad \text{id2} \in \text{USER.consults(users(s)(id1)))} \Rightarrow \\
\quad\quad\quad \text{id2} \in \text{copys(s))}
\end{array}
$$

This kind of predicates for checking existence take slightly different forms depending on the association end multiplicities and where the class containers are.

When the association is bidirectional, besides checking existence as before, it is also necessary to check binavigability. By binavigability we means: when from an object of class C1 we get –through the association– a set of objects of class C2 and, then if we navigate again through the association in the opposite direction from all these objects of class C2 to C1 we get again the same starting object in C1. The same control must be achieved when we navigate from C2 to C1. Therefore, two predicates must be generated for that.

For instance, the association "consults" between classes "Copy" and "User" that is shown in figure 5 is bidirectional. If we want to check binavigability -i.e. to check for all the copies consulted by a given user that each copy has been actually consulted by this user- we might include in our specification the following two predicates:

$$
\begin{array}{l}
(\forall\ \text{id1} : \text{User_Id},\ \text{id2} : \text{Copy_Id} \bullet \\
\quad (\text{id1} \in \textbf{dom}\ \text{users(s)} \wedge \\
\quad\quad \text{id2} \in \text{USER.consults(users(s)(id1)))} \Rightarrow \\
\quad\quad\ (\text{id2} \in \text{copys(s)} \wedge \\
\quad\quad\quad \text{an_User(id1)} = \text{COPY.consults(copys(s)(id2))))}
\end{array}
$$

$$
\begin{array}{l}
(\forall\ \text{id1} : \text{Copy_Id},\ \text{id2} : \text{User_Id} \bullet \\
\quad (\text{id1} \in \textbf{dom}\ \text{copys(s)} \wedge \\
\quad\quad \text{an_User(id2)} = \text{COPY.consults(copys(s)(id1)))} \Rightarrow \\
\quad\quad\ (\text{id2} \in \text{users(s)} \wedge \\
\quad\quad\quad \text{id1} \in \text{USER.consults(users(s)(id2))))}
\end{array}
$$

Note that, in general, a predicate for checking existence has the form:

$$\forall \text{ x: X, y: Y} \bullet (P(x) \wedge Q(x,y)) \Rightarrow R(y)$$

and the corresponding one for checking binavigability has the form:

$$\forall \text{ x: X, y: Y} \bullet (P(x) \wedge Q(x,y)) \Rightarrow (R(y) \wedge S(x,y))$$

It can be shown that binavigability implies existence. Therefore, when we have a bidirectional association, the two corresponding predicates for existence are not given since they are implied from the binavigability predicates.

In appendix F the variations for building existence and binavigability predicates are given.

When we have an association class, the corresponding predicates for existence or binavigability must be generated for the associations resulting from transformations proposed in section 5.2.1.

6 The Tool

Currently there are several commercial or freely available UML-based graphical tools. On the other hand, XMI is the more commonly used format in which these tools store the information about the models. Based on these facts, our application uses as input a class diagram produced by a tool which saves its diagrams in the XMI format.

XMI is an acronym for "**X**ML **M**etadata **I**nterchange" and XML stands for "e**X**tensible **M**arkup **L**anguge". Although XMI is a technology from the OMG (**O**bject **M**anagement **G**roup) [1], it is based on the XML standard from the W3C (the World Wide Web Consortium) [2]. XMI is a family of XML. Unfortunately, there exists several versions for the UML XMI DTD (**D**ocument **T**ype **D**efinition). Our tool is compatible with the XML files compliant to the XMI DTD version 1.0. As it is shown in the figure 22, it takes as input an XML file produced by a UML-based graphical tool, where a class diagram has been stored, it parses the XML file and, if the input is syntactically correct, translates the class diagram to an RSL specification based on our previously proposed semantics.

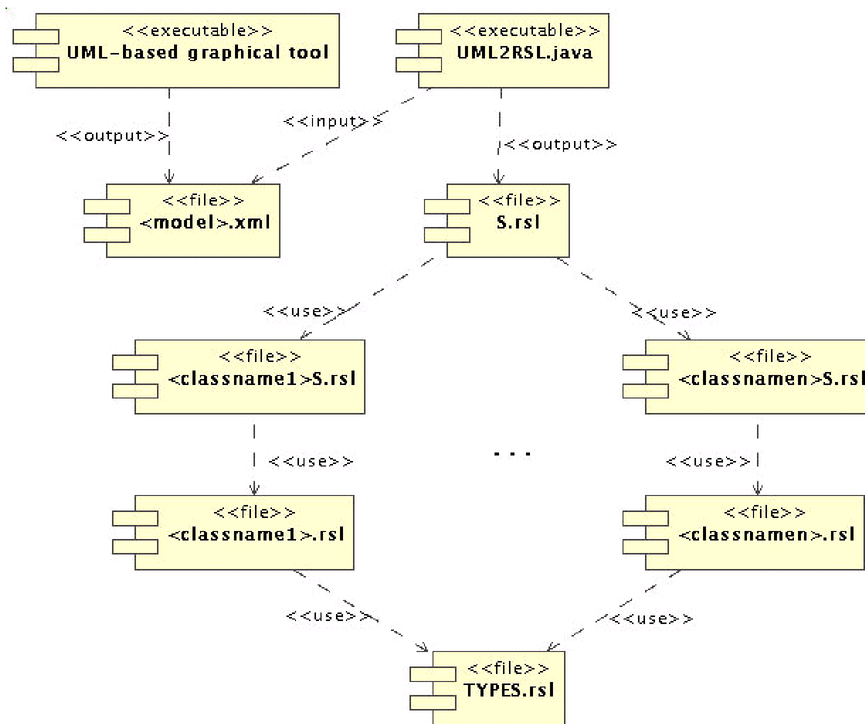

Figure 22: Component diagram

The whole produced RSL specification consists of several RSL files. The top level module "S.rsl" –that corresponds to the specification of the model represented by the class diagram – uses a set of modules produced for the specification of each class that has been depicted in the class

diagram (these modules receive as name the corresponding class name in upper case, followed by "S"). Each one, in turn, uses a lower level module where the specification for one object of the corresponding class is given, and each one of these lower level modules uses, in turn, the module "TYPES.rsl" where all the abstract types present in the diagram are defined.

The tool has been developed in Java, making use of a commonly used API (Application Program Interface) for XML processors: the **D**ocument **O**bject **M**odel (DOM) API [19] [13]. In DOM, when an XML document is parsed it is represented as a tree. DOM provides a set of APIs to access and manipulate these nodes in the DOM tree. We used the DOM API contained in the org.apache.xerces.parsers.DOMParser package, which can be downloaded from http://www.alphaworks.ibm.com/tech/xml4j.

7 Concluding Remarks

Summary. In the present work we have explored the use of RSL to formalize several elements present in UML class diagrams. Some of these constructs are typically used during the creation of conceptual models, while others appear commonly during the design phase. We have given the semantics for all the basic constructs used for building problem-oriented class diagrams as well as for several of those used during the design. Also a discussion and specification of class diagram well-formedness rules have been presented.

The analysis of the semantics of UML class diagrams and its formal specification in RSL have allowed us to abstract a series of templates, which have guided the implementation of a translator tool.

The definition of a mechanism to give the formal foundations of UML class diagrams has enhanced the precision of UML as a specification language, it has opened the possibility of carrying out rigorous analysis on the model and has allowed the creation of initial RSL specifications feasible for further refinements.

Related Work. [16] presents a formal mapping between UML class diagrams and Object-Z achieved at a meta level. [9] also presents the semantics for some UML class diagram constructs, but using Z. The work is mainly centered on the discussion of alternative interpretations of the UML, and on examples of its use for rigorous analysis. In [7] an algebraic model for object-orientation concepts is presented. It is based on the use of Rumbaugh's Object Modeling Technique (OMT) and the specification language O-SLANG. [21] proposes the use of a subset of UML class diagrams to define a particular domain of refinement relations, which is formalized using Z. [20] presents an approach to transform OMT class diagrams and state diagrams into B specifications. The approach is based –as our work– in the use of templates, however no automated tool is reported. No one considers scope of methods and attributes, attribute changeability, and template classes.

Future Work. As we have seen, in general, our semantics for a class diagram results in a four layer RSL module structure, where the top level module has assigned many responsibilities. This drawback is due to the fact that UML allows the creation of loops between associations and RSL encourages the structured design and consequently does not allow circular dependencies of modules.

UML class diagrams built for modeling the domain of the problem typically present this shape. Since a domain model is not a software model, module dependency and delegations are not issues that matter. However, when we build a design class diagram they should be considered. Therefore, whenever we have a good UML design model, we should get a good one in RSL too.

In general, a good design model should avoid the use of bidirectional associations, and the depen-

dencies between modules should form directed acyclic graphs. Bidirectional associations make maintenance more complex and lead to an interdependency between the classes and consequently to highly coupled systems.

In order to improve the RSL structure, initial efforts will focus on finding a way to obtain a better RSL system architecture based in the interdependencies between the classes or in complementary UML diagrams.

Future work includes also the creation of mappings between RSL and other UML diagrams, mainly focused on the formalization of behavioral diagrams paying attention to the methodological aspects concerned with the integration of both notations.

Acknowledgments

I would like to thank UNU/IIST for the support and help given, and specially my adviser Chris George for his guidance and support during my stay in Macau, where this work was developed.

Thanks to UNSL that also made possible my stay at UNU/IIST, and to my son Juan for his patience.

Appendices

A Templates for a Class

The text that is fixed in the template is either in **bold style** (RSL keywords) or in normal style, while text in *italic* is replaced automatically using the corresponding names in the UML diagram. The indices are used to reference a particular element between a set of elements, and the references between braces on the right margin indicate conditions that must be satisfied to generate the preceding text.

In order to give the templates for a class, let us assume that the class has m instance-scope attributes ($0 \leq m$), b class-scope attributes ($0 \leq b$), r instance-scope operations ($0 \leq r$), s class-scope operations ($0 \leq s$) and a navigable associations ($0 \leq a$).

```
CLASSNAME
object CLASSNAMES :
  with TYPES in
  class
    type
      Classname = CLASSNAME.Classname,
      Classnames = Classname_Id →m Classname

    value
      empty_Classname: Classnames = [ ],

      add_Classname: Classname_Id × Classname × Classnames →~ Classnames
      add_Classname(id, o, c) ≡ c † [id ↦ o]
        pre ∼ isin_Classname(id, c),

      del_Classname: Classname_Id × Classnames →~ Classnames
      del_Classname(id, c) ≡ c \ {id}
        pre isin_Classname(id, c),

      isin_Classname: Classname_Id × Classnames → Bool
      isin_Classname(id, c) ≡ id ∈ dom c,

      get_Classname: Classname_Id × Classnames →~ Classname
      get_Classname(id, c) ≡ c(id)
        pre isin_Classname(id, c),

      update_Classname: Classname_Id × Classname × Classnames →~ Classnames
      update_Classname(id, o, c) ≡ c † [id ↦ o]
        pre isin_Classname(id, c),

      operationName_i: ParameterType_i1 × ... ×
```

$ParameterType_{it} \times Classnames \rightarrow ResulType_i,$ {1}

consistent: $Classnames \rightarrow$ **Bool**
consistent(c) $\equiv$
  ($\forall$ id: $Classname$_Id •
    id $\in$ **dom** c $\Rightarrow$ CLASSNAME.consistent(c(id)))
  $\wedge$ $multiplicity_checking$(c) {2}
  $\wedge$ $class_scoped_attribute_checking_i$(c) {3}

end

$\{1\} : 0 < i \leq s \wedge numberOfParameters(Operation_i) = t$
$\{2\} : multiplicity(c) > 1 \wedge multiplicity(c) \neq *$
$\{3\} : 0 < i \leq b$

$multiplicity_checking$(c) is the predicate for checking the multiplicity of the class, and $class_scoped_attribute_checking_i$(c) are the predicates generated for each class-scoped attribute in order to check that all the instances currently in the class have the same attribute value.

TYPES
object $CLASSNAME$:
  with TYPES **in**
  hide $operationName_i$ **in** {4}
  class
    type
      $Classname$
    value
      $attributeName_i$: $Classname \rightarrow AttributeType_i$, {5}

      update_$attributeName_i$: $AttributeType_i \times Classname \stackrel{\sim}{\rightarrow} Classname$
      update_$attributeName_i$(at, o) **as** o′ **post** $attributeName_i$(o′) = at
        pre preupdate_$attributeName_i$(at, o),

      preupdate_$attributeName_i$: $AttributeType_1 \times Classname \rightarrow$ **Bool**, {5}

      $asociationName_i$: $Classname \rightarrow AssociationType_i$, {6}

      update_$associationName_i$: $AssociationType_i \times Classname \stackrel{\sim}{\rightarrow} Classname$
      update_$associationName_i$(a, o) **as** o′ **post** $associationName_i$(o′) = a
        pre preupdate_$associationName_i$(a, o),

      preupdate_$associationName_i$: $AssociationType_i \times Classname \rightarrow$ **Bool**, {6}

$operationName_i$: $ParameterType_{i1} \times ... \times$
$ParameterType_{it} \times Classname \rightarrow ResulType_i$, {7}

consistent: $Classname \rightarrow$ **Bool**
consistent(o) $\equiv$
attribute_multiplicity_checking$(Attribute_i)$ {8}
$\wedge$ *association_multiplicity_checking*$(Association_i)$ {9}

end

$\{4\} : 0 < i \leq r \wedge Operation_i$ is an abstract operation
$\{5\} : 0 < i \leq m + b$
$\{6\} : 0 < i \leq a$
$\{7\} : 0 < i \leq r \wedge numberOfParameters(Operation_i) = t$
$\{8\} : 0 < i \leq m \wedge multiplicity(Attribute_i) > 1 \wedge multiplicity(Attribute_i) \neq *$
$\{9\} : 0 < i \leq \wedge multiplicity(Association_i) > 1 \wedge multiplicity(Association_i) \neq *$

B Templates for a Subclass

Let us assume that the subclass has m instance-scope attributes ($0 \leq m$), b class-scope attributes ($0 \leq b$), r instance-scope operations ($0 \leq r$), s class-scope operations ($0 \leq s$) and a navigables associations ($0 \leq a$).

SUPERCLASSNAMES, *CLASSNAME*
object *CLASSNAMES* :
  with TYPES **in**
  class
    type
      Classname = *CLASSNAME.Classname*,
      Classnames =
        {| super : *SUPERCLASSNAMES.Superclassnames*•
          (∀ id : *Superclassname*_Id •
            id ∈ **dom** super ⇒
            CLASSNAME.is_a_*Classname*(super(id))) |}

    value
      empty_*Classname*: *Classnames* = [ ],

      add_*Classname*: *Classname*_Id × *Classname* × *Classnames* $\stackrel{\sim}{\rightarrow}$ *Classnames*
      add_*Classname*(id, o, c) ≡ c † [id ↦ o]
        pre ∼ isin_*Classname*(id, c),

      del_*Classname*: *Classname*_Id × *Classnames* $\stackrel{\sim}{\rightarrow}$ *Classnames*
      del_*Classname*(id, c) ≡ c \ {id}
        pre isin_*Classname*(id, c),

      isin_*Classname*: *Classname*_Id × *Classnames* → **Bool**
      isin_*Classname*(id, c) ≡ id ∈ **dom** c,

      get_*Classname*: *Classname*_Id × *Classnames* $\stackrel{\sim}{\rightarrow}$ *Classname*
      get_*Classname*(id, c) ≡ c(id)
        pre isin_*Classname*(id, c),

      update_*Classname*: *Classname*_Id × *Classname* × *Classnames* $\stackrel{\sim}{\rightarrow}$ *Classnames*
      update_*Classname*(id, o, c) ≡ c † [id ↦ o]
        pre isin_*Classname*(id, c),

      $operationName_i$: $ParameterType_{i1}$ × ... ×
        $ParameterType_{it}$ × *Classnames* → $ResulType_i$, {1}

consistent: *Classnames* → **Bool**
consistent(c) ≡
(∀ id: *Classname*_Id •
id ∈ **dom** c ⇒ CLASSNAME.consistent(c(id)))
∧ *multiplicity_checking*(c) {2}
∧ *class_scoped_attribute_checking*$_i$(c) {3}

end

{1} : $0 < i \leq s \wedge numberOfParameters(Operation_i) = t$
{2} : $multiplicity(c) > 1 \wedge multiplicity(c) \neq *$
{3} : $0 < i \leq b$

multiplicity_checking(c) is the predicate for checking the multiplicity of the class, and *class_scoped_attribute_checking*$_i$(c) are the predicates generated for each class-scoped attribute in order to check that all the instances currently in the class have the same attribute value.

SUPERCLASSNAME
object *CLASSNAME* :
with TYPES **in**
hide *operationName*$_i$ **in** {4}
class
type
Superclassname = *SUPERCLASSNAME.Superclassname*,
Classname = {|o: *Superclassname* • is_a_*Classname*(o)|}

value
attributeName$_i$: *Classname* → *AttributeType*$_i$, {5}

update_*attributeName*$_i$: *AttributeType*$_i$ × *Classname* $\xrightarrow{\sim}$ *Classname*
update_*attributeName*$_i$(at, o) **as** o′ **post** *attributeName*$_i$(o′) = at
pre preupdate_*attributeName*$_i$

preupdate_*attributeName*$_i$: *AttributeType*$_i$ {5}

asociationName$_i$: *Classname* → *AssociationType*$_i$, {6}

update_*associationName*$_i$: *AssociationType*$_i$ × *Classname* $\xrightarrow{\sim}$ *Classname*
update_*associationName*$_i$(a, o) **as** o′ **post** *associationName*$_i$(o′) = a
pre preupdate_*associationName*$_i$(a, o),

preupdate_$associationName_i$: $AssociationType_i \times Classname \rightarrow$ **Bool**, {6}

$operationName_i$: $ParameterType_{i1} \times ... \times$
$ParameterType_{it} \times Classname \rightarrow Resultype_i$, {7}

is_a_$Classname$: $Superclassname \rightarrow$ **Bool**,

consistent: $Classname \rightarrow$ **Bool**
consistent(o) $\equiv$
$SUPERCLASSNAME$.consistent(o)
$\wedge$ *attribute_multiplicity_checking(Attribute$_i$)* {8}
$\wedge$ *association_multiplicity_checking(Association$_i$)* {9}

end

{4}: $0 < i \leq r \wedge Operation_i$ is an abstract operation.
{5}: $0 < i \leq m + b$
{6}: $0 < i \leq a$
{7}: $0 < i \leq r \wedge numberOfParameters(Operation_i) = t$
{8}: $0 < i \leq m \wedge multiplicity(Attribute_i) > 1 \wedge multiplicity(Attribute_i) \neq *$
{9}: $0 < i \leq a \wedge multiplicity(Association_i) > 1 \wedge multiplicity(Association_i) \neq *$

C Templates for Leaf Classes

The only difference between the templates for leaf classes and that given in the appendix A is the specification for an object of the leaf class.

Let us assume that the class has m instance-scope attributes $(0 \leq m)$, b class-scope attributes $(0 \leq b)$, r instance-scope operations $(0 \leq r)$, s class-scope operations $(0 \leq s)$ and a navigable associations $(0 \leq a)$. Note that when $m + a$ is equal to 0 the template given in the appendix A is used.

TYPES
object CLASSNAME :
  with TYPES **in**
  hide $operationName_i$ **in** {1}
  class
    type
      Classname ::
        $attributeName_i$: $AttributeType_i \leftrightarrow$ replace_$attributeName_i$ {2}
        $asociationName_i$: $AssociationType_i \leftrightarrow$ replace_$associationName_i$ {3}

    value
      update_$attributeName_i$: $AttributeType_i \times Classname \stackrel{\sim}{\rightarrow} Classname$
      update_$attributeName_i$(at, o) $\equiv$ replace_$attributeName_i$(at, o)
        pre preupdate_$attributeName_i$(at, o),

      preupdate_$attributeName_i$: $AttributeType_i$

      update_$associationName_i$: $AssociationType_i \times Classname \stackrel{\sim}{\rightarrow} Classname$
      update_$associationName_i$(a, o) $\equiv$ replace_$associationName_i$(a, o)
        pre preupdate_$associationName_i$(a, o),

      preupdate_$associationName_i$: $AssociationType_i \times Classname \rightarrow$ **Bool**, {3}

      $operationName_i$: $ParameterType_{i1} \times ... \times$
        $ParameterType_{it} \times Classname \rightarrow ResulType_i$, {4}

      consistent: $Classname \rightarrow$ **Bool**
      consistent(c) $\equiv$
        attribute_multiplicity_checking($Attribute_i$) {5}
        $\wedge$ *association_multiplicity_checking*($Association_i$) {6}

end

$\{1\}$: $0 < i \leq r \wedge Operation_i$ is an abstract operation.
$\{2\}$: $0 < i \leq m + b$
$\{3\}$: $0 < i \leq a$
$\{4\}$: $0 < i \leq r \wedge numberOfParameters(Operation_i) = t$
$\{5\}$: $0 < i \leq m \wedge multiplicity(Attribute_i) > 1 \wedge multiplicity(Attribute_i) \neq *$
$\{6\}$: $0 < i \leq a \wedge multiplicity(Association_i) > 1 \wedge multiplicity(Association_i) \neq *$

D Template for a Class Diagram

Let us assume that the class diagram has n concrete classes with (n > 0). By concrete we mean those classes that are not template classes.

$CLASSNAME_i$S {1}
object S :
  with TYPES **in**
  class
    type
      Sys ::
        $classname_i$s: $CLASSNAME_i$S.$Classname_i$s ↔ replace_$Classname_i$s {2}

    value
      $classname_i$s: Sys → $CLASSNAME_i$S.$Classname_i$s
      $classname_i$s(s) ≡
          $superclassname_i$s(s) /
            {id |
             id : *Classname*_Id •
               id ∈ **dom** *superclassnames*(s) ∧
               CLASSNAME.is_a_*Classname*(*superclassnames*(s)(id))},

      update_$classname_i$s: $CLASSNAME_i$S.$Classname_i$s × Sys $\stackrel{\sim}{\rightarrow}$ Sys
      update_$classname_i$s(c, s) ≡ update_$superclassname_i$s(c, s)
        pre preupdate_$classname_i$s(c, s),

      preupdate_$classname_i$s: $CLASSNAME_i$S.$Classname_i$s × Sys → **Bool**, {3}

      update_$classname_i$s: $CLASSNAME_i$S.$Classname_i$s × Sys $\stackrel{\sim}{\rightarrow}$ Sys
      update_$classname_i$s(c, s) ≡ replace_$Classname_i$s(c, s)
        pre preupdate_$classname_i$s(c, s),

      preupdate_$classname_i$s: $CLASSNAME_i$S.$Classname_i$s × Sys → **Bool**, {4}

      del_$Classname_i$: *Classname*_Id × Sys $\stackrel{\sim}{\rightarrow}$ Sys
      del_$Classname_i$(id, s) **as** s″ **post**
        (∃
          s′ : Sys, new_whole : $CLASSNAME_i$S.$Classname_i$s;
          new_parts: $PARTCLASSNAME_j$S.*PartClassnames*,
          parts: *PartClassname*_Id-**set**
          •
          parts =
            $CLASSNAME_i$.$composition_j$($classname_i$s(s)(id)) ∧ {1.1}
          new_parts = *partClassnames*(s) \ parts ∧

s′ = update_*partClassnames*s(new_parts, s) ∧
new_whole =
$CLASSNAME_i$S.del_$Classname_i$s(id, $classname_i$s(s′)) ∧
s″ = update_$Classname_i$s(new_whole, s))
pre can_del_$Classname_i$(id, s), {1}

can_del_$Classname_i$: *Classname*_Id × Sys → **Bool**
can_del_$Classname_i$(id, s) ≡
$CLASSNAME_i$S.isin_$Classname_i$(id, $classname_i$s(s)) ∧
(∃
s′ : Sys, new_whole : $CLASSNAME_i$S.$Classname_i$s;
new_parts: $PARTCLASSNAME_j$S.*PartClassnames*,
parts: *PartClassname*_Id-**set**
•
parts =
$CLASSNAME_i$.$composition_j$($classname_i$s(s)(id)) ∧ {1.1}
new_parts = *partClassnames*(s) \ parts ∧
preupdate_*partClassnames*(new_parts, s) ∧
s′ = update_*partClassnames*(new_parts, s) ∧
new_whole =
$CLASSNAME_i$S.del_$Classname_i$s(id, $classname_i$s(s′)) ∧
preupdate_$Classname_i$s(new_whole, s)),

$operationName_{ij}$_in_$Classname_i$: $ParameterType_{ij1}$ × ... ×
$ParameterType_{ijt}$ × $Classname_i$ × Sys → $ResulType_{ij}$, {5}

consistent: Sys → **Bool**
consistent(s) ≡
$CLASSNAME_i$S.consistent($classname_i$s(s)) {1}
∧ *binavegability_checkings*
∧ *existence_checkings*
∧ *abstract_class_checkings*

axiom
∀ s: Sys, c: $CLASSNAME_i$S.$Classname_i$s• update_$classname_i$s(c,s) **as** s′
post consistent(s′) ∧
$frozenCheckings_i$ ∧
$addOnlyCheckings_i$
pre consistent(s) ∧
preupdate_$classname_i$s(c, s) {1}

end

$\{1\}$: $0 < i \leq n$
$\{1.1\}$: $0 < j \leq numberOfParts(Class_i)$
$\{2\}$: $0 < i \leq n \wedge Class_i$ is not a subclass.
$\{3\}$: $Class_i$ is a subclass.
$\{4\}$: $Class_i$ is not a subclass.
$\{5\}$: $0 < i \leq n \wedge 0 < j \leq numberOfOperations(Class_i) \wedge Operation_{ij}$ is not abstract.

The existence and binavigabilty checkings correspond to all the associations in the class diagram. The *abstract_class_checkings* are generated when there are abstract classes in the class diagram.

The checkings in the post conditions are generated only when the class has attributes and/or associations having {frozen} or {addOnly} property as explained in section 5.8.

E Templates for Parameterized Classes

In order to give the templates for a template class and for an instantiated class, let us assume that the template class has m instance-scope attributes ($0 \leq m$), b class-scope attributes ($0 \leq b$), r instance-scope operations ($0 \leq r$), s class-scope operations ($0 \leq s$), a navigable associations ($0 \leq a$), $p1$ typed formal parameters ($0 \leq p1$) and $p2$ untyped formal parameters ($0 \leq p2$) with $p1 + p2 \neq 0$.

Parameterized classes cannot have instances. Only instantiated classes can. The template for the class container of an instantiated class is the same that for any other concrete class.

E.1 Template for Template Classes

TYPES,
scheme *TEMPLATECLASSNAME_*(
  FPAR :
    with TYPES **in**
    class
      type
        $FormalParameterName_1 ... FormalParameterName_{p2}$

      value
        $formalParameterName_i$: $FormalParameterType_i$, {1}
    end) =
  with TYPES **in**
  class
    type *TemplateClassname*

    type $FormalParameterName_i$s = $FormalParameterName_i$-**set** {2}

    type Optional_$FormalParameterName_i$s =
        no_$FormalParameterName_i$ |
        a_$FormalParameterName_i$(Id:$FormalParameterName_i$) {3}
    value
      $attributeName_i$: *TemplateClassname* $\rightarrow$ $AttributeType_i$, {4}

      update_$attributeName_i$: $AttributeType_i$ $\times$ *TemplateClassname* $\stackrel{\sim}{\rightarrow}$ *TemplateClassname*
      update_$attributeName_i$(at, o) **as** o′ **post** $attributeName_i$(o′) = at
        pre preupdate_$attributeName_i$(at, o),

      preupdate_$attributeName_i$: $AttributeType_i$ $\times$ *TemplateClassname* $\rightarrow$ **Bool**, {4}

$associationName_i$: $TemplateClassname \rightarrow AssociationType_i$, {5}

update_$associationName_i$: $AssociationType_i \times TemplateClassname \stackrel{\sim}{\rightarrow} TemplateClassname$
update_$associationName_i$(a, o) **as** o′ **post** $associationName_i$(o′) = a
pre preupdate_$associationName_i$(a, o),

preupdate_$associationName_i$: $AssociationType_i \times TemplateClassname \rightarrow$ **Bool**, {5}

$operationName_i$: $ParameterType_{i1} \times ... \times$
$ParameterType_{it} \times TemplateClassname \rightarrow ResulType_i$, {6}

consistent: $TemplateClassname \rightarrow$ **Bool**
consistent(o) ≡
attribute_multiplicity_cheking(Attribute$_i$) {7}
∧ *association_multiplicity_cheking(Association$_i$)* {8}

end

{1}: $1 \leq i \leq p1$
{2}: $0 < i \leq p2 \wedge (\exists_j : Nat \bullet 0 <_j < m \wedge type(Attribute_j) = FormalParameterName_i \wedge Multiplicity(Attribute_j) > 1)$
{3}: $0 < i \leq p2 \wedge (\exists_j : Nat \bullet 0 <_j < m \wedge type(Attribute_j) = FormalParameterName_i \wedge (Multiplicity(Attribute_j) = 0 \vee Multiplicity(Attribute_j) = 1))$
{4}: $0 < i \leq m + b$
{5}: $0 < i \leq a$
{6}: $0 < i \leq r \wedge numberOfParameters(Operation_i) = t$
{7}: $0 < i \leq m \wedge multiplicity(Attribute_i) > 1 \wedge multiplicity(Attribute_i) \neq *)$
{8}: $0 < i \leq a \wedge multiplicity(Association_i) > 1 \wedge multiplicity(Association_i) \neq *$

E.2 Templates for Instantiated Classes

*TEMPLATECLASSNAME_*
scheme *INSTANTIATEDCLASSNAME_* =
with TYPES **in**
use is_a_*InstantiatedClassname* **for** is_a_*TemplateClassname* **in** {9}

extend
class
object
APAR_*InstantiatedClass* :
class

type
$FormalParameterName_i = ActualParameter_i,$ $\{10\}$
value
$formalParameterName_i : FormalParameterType_i = ActualParameter_i,$ $\{11\}$

end
end
with extend *TEMPLATECLASSNAME_*(APAR_*InstantiatedClass*) **with class**
type *InstantiatedClass* = *TemplateClassname* **end**

$\{9\}$: *TemplateClass* is a subclass.
$\{10\}$: $1 \leq i \leq p2$
$\{11\}$: $1 \leq i \leq p1$

*INSTANTIATEDCLASSNAME_*
object *INSTANTIATEDCLASSNAME*: *INSTANTIATEDCLASSNAME_*

F Templates for constraints

F.1 Class multiplicities

Class multiplicity checks are placed into the function "consistent" that corresponds to the module that have the specification for the class. When the class multiplicity is "$N..*$", for a given $N > 0$, we have that the template takes the following form:

consistent: *Classnames* $\rightarrow$ **Bool**
consistent(c) $\equiv$ .... $\wedge$ **card dom** c $\geq$ N

For multiplicity "$N..M$", with $M > N \geq 0$, the predicate will be generated from the template:

card dom c $\geq$ N $\wedge$ **card dom** c $\leq$ M

If multiplicity is a given fixed number N, $N >= 0$, the corresponding predicate is:

card dom c = N

Any other class multiplicity does not generate a constraint in "consistent".

F.2 Association end multiplicities

To give the templates corresponding to the end multiplicities, first we need to present the templates for translating associations.

Let us suppose that we have an association embedded in class c_i that allows to navigate to class c_j, i.e. in class c_i we have:

associationName: $Classname_i \rightarrow$ *Associationtype*

Depending on the association end multiplicity at c_j, the type of the association -denoted *Associationtype*- will be replaced either by the type corresponding to an object identifier of class c_j, to a set of object identifiers of class c_j, or to a variant type.

In the former case, when the multiplicity is "1" we have that *Asociationtype* is replace by "$Classname_j$_Id", defined in "TYPES". For a multiplicity of "0..1", we define in the module "TYPES" a variant type:

$$\text{Optional_}Classname_i == \\ \quad \text{no_}Classname_i \mid \text{a_}Classname_i(\text{id} : Classname_i\text{_Id})$$

Then, *AsociationType* is replaced by Optional_$Classname_i$.

For any other multiplicity, *Asociationtype* is replaced by "$Classname_j$_Id-set".

For each navigable association end –when a multiplicity greater than one and different to "*" is attached to the end– a multiplicity constraint is generated. Consequently, given an association between class c_i and class c_j, one of three different predicates may be generated according to the association end's multiplicity. The analysis is done for association end at c_j.

When the multiplicity at the association end corresponding to c_j is "$N..*$" for a given $N > 0$, a predicate in the function "consistent" of the module that holds the specification of the class c_i is generated. Assuming that o –an object of the class c_i– is the parameter of the function "consistent", the predicate is given by of the following template:

$$\textbf{card}\ associationName(\text{o}) \geq \text{N}$$

For multiplicity "$N..M$", with N and M given and $M \geq N > 0$, the predicate template is:

$$\textbf{card}\ associationName(o) \geq \text{N} \wedge \textbf{card}\ associationName(\text{o}) \leq \text{M}$$

And finally, when the multiplicity is given by a fixed number N, with $N > 1$, the corresponding predicate template is:

$$\textbf{card}\ asociationName(\text{o}) = \text{N}$$

F.3 Attribute multiplicities

In order to give the templates for attribute multiplicities let assume that the RSL observer to obtain the attribute is:

value
attributeName : *Classname* → *AttributeType*

When a template class is translated and the attribute type corresponds to a formal parameter, *AttributeType* is replaced by the formal parameter if the multiplicity is equal to one. When the multiplicity is equal to "0..1" then is replaced by an optional type which is defined in the current module:

type
Optional_*FormalParametername* ==
no_*FormalParametername* |
a_*FormalParametername*(Id: FPAR.*FormalParametername*)

For any other multiplicity, *Attributetype* is replaced by the type *FormalParameternames*, which is also defined locally:

type
FormalParameternames = FPAR.*FormalParametername*-**set**

When *Attributetype* is not a formal parameter, it receive the same treatment as an attribute that belongs to a concrete class. That is, all those attributes whose type is a class sort are interpreted as one direction composition relationships (see sections 5.6 and 5.8.1). In any other case, they are specified as an object identifier, a variant type of object identifier type or a set of object identifiers.

When the attribute has a multiplicity greater than 1 and different to "*", the attribute is specified as a set whose cardinality is restricted by its multiplicity. Consequently, a predicate in the function "consistent" –defined for an object o of the class– is generated according to the attribute multiplicity.

If the attribute multiplicity is "$N..*$", for a given $N > 0$, we will have in consistent(o):

card *attributeName*(o) ≥ N

For a multiplicity "$N..M$", with N and M given and $M > N > 0$, the predicate template is:

card *attributeName*(o) ≥ N ∧ **card** *attributeName*(o) ≤ M

If the multiplicity is a given fixed number N, with $N > 1$, the corresponding predicate template is:

$$\textbf{card}\ \mathit{attributeName}(\mathrm{o}) = \mathrm{N}$$

Any other attribute multiplicity does not generate a constraint in the function "consistent".

F.4 Class-scope attributes

The templates used for generating the predicates for checking that an attribute is class-scoped are always placed in the module that holds the specification of the class. If c is the class that is the parameter of the function "consistent", then we have:

$$\begin{array}{l}(\forall\ \mathrm{id1},\ \mathrm{id2} :\ \mathrm{TYPES}.\mathit{Classname}_\mathrm{Id} \bullet \\ \quad (\mathrm{id1} \in \textbf{dom}\ \mathrm{c} \wedge \mathrm{id2} \in \textbf{dom}\ \mathrm{c}) \Rightarrow \\ \quad\quad (\mathit{CLASSNAME.attributeName}(\mathrm{c}(\mathrm{id1})) = \\ \quad\quad\quad \mathit{CLASSNAME.attributeName}(\mathrm{c}(\mathrm{id2}))))\end{array}$$

F.5 Attribute and association-end properties

To check that the attribute properties {frozen} and {addonly} hold in the system it is necessary to consider the system state before and after a change has taken place. For this reason, the predicates used for checking these properties are not used inside a function "consistent", but they are used in the post conditions of the top-level axioms.

When a given class *Classname* has one or more frozen attributes, we define a boolean function "frozenAtts_in_*Classname*" according to the template given below. This function is used inside the post condition of the top level axioms used for checking system consistency. For instance, if *attributeName* is the name of one frozen attribute in class *Classname*, we have:

$$\begin{array}{l}\mathrm{frozenAtts_in_}\mathit{Classname}\text{: Sys} \times \text{Sys} \rightarrow \textbf{Bool} \\ \mathrm{frozenAtts_in_}\mathit{Classname}(\mathrm{s}', \mathrm{s}) \equiv \\ \quad (\forall\ \mathrm{id} :\ \mathrm{TYPES}.\mathit{Classname}_\mathrm{Id} \bullet \\ \quad\quad (\mathrm{id} \in \textbf{dom}\ \mathit{classnames}(\mathrm{s}) \wedge \\ \quad\quad\ \ \mathrm{id} \in \textbf{dom}\ \mathit{classnames}(\mathrm{s}')) \Rightarrow \\ \quad\quad\quad \mathit{CLASSNAME.attributeName}(\mathit{classNames}(\mathrm{s})(\mathrm{id})) = \\ \quad\quad\quad\quad \mathit{CLASSNAME.attributeName}(\mathit{classnames}(\mathrm{s}')(\mathrm{id})))\end{array}$$

Similar templates are used for property {addonly}. On the basis of the previous one, the function name is changed from "frozenAtts_in_*Classname*" to "addOnlyAtts_in_*Classname*" and the equality (=) between sets is changed to inclusion ($\subseteq$). The same templates are used for association-end properties by just changing *atributeName* to *associationName*.

F.6 Abstract classes

Given an abstract class *Classname* with n subclasses $Subclassname_i$ $(0 \leq i \geq n)$, a constraint based in the template given below is generated in the function "consistent" of the module that holds the specification of the whole system.

consistent: Sys $\rightarrow$ **Bool**
consistent(s) $\equiv$ ...$\wedge$
    dom *classname*
        = **dom** $subclassname_1$ $\cup$ ... $\cup$ **dom** $subclassname_n$ {1}
        = {} {2}

{1} : $0 < n$
{2} : $n = 0$

F.7 Existence constraints

Existence constraints are used for checking in the class container the existence of those objects that are obtained from the functions used to specify a given association. Let assume the existence of a one-direction association from a class $Classname_i$ to another class $Classname_j$. The templates for checking existence are based on:

($\forall$ id1: TYPES.$Classname_i$_Id,
    id2: TYPES.$Classname_j$_Id •
    (id1 $\in$ **dom** $classname_i$s(s) $\wedge$
      id2 $\in$ $CLASSNAME_i.associationName(classname_i$s(id1))) $\Rightarrow$
      id2 $\in$ **dom** $classname_j$s(s))

where s denotates an instance of the system.

When the multiplicity is "0..1", the predicate

$$\text{id2} \in \mathit{CLASSNAME}_i.\mathit{associationName}(\mathit{classname}_i\text{s}(\text{id1}))$$

in the antecedent, is changed to:

$$\text{TYPES.a_}\mathit{Classname}(\text{id2}) = \mathit{CLASSNAME}_i.\mathit{associationName}(\mathit{classname}_i\text{s}(\text{id1}))$$

For multiplicity equal to "1..1":

$$\text{id2} = \mathit{CLASSNAME}_i.\mathit{associationName}(\mathit{classname}_i\text{s}(\text{id1}))$$

F.8 Bi-navigability constraints

When an association is bidirectional, i.e. it is possible to navigate in both directions, it is necessary to generate two predicates for checking bi-navigability. Below we present the templates for checking bi-navigability for a given association *associationName* between classes $\mathit{Classname}_i$ and $\mathit{Classname}_j$. Only one of them is given. The other is equal to the first one but with $_i$ and $_j$ interchanged.

$$
\begin{aligned}
&(\forall\ \text{id1} : \text{TYPES.}\mathit{Classname}_i\text{_Id}, \\
&\quad \text{id2} : \text{TYPES.}\mathit{Classname}_j\text{_Id} \bullet \\
&\quad (\text{id1} \in \mathbf{dom}\ \mathit{classname}_i\text{s(s)} \land \\
&\quad \text{id2} \in \mathit{CLASSNAME}_i.\mathit{associationName}(\mathit{classname}_i\text{s}(\text{id1}))) \Rightarrow \\
&\qquad \text{id2} \in \mathbf{dom}\ \text{classnames}_j\text{(s)} \land \\
&\qquad \text{id1} \in \mathit{CLASSNAME}_j.\mathit{associationName}(\mathit{classname}_j\text{s}(\text{id2})))
\end{aligned}
$$

As existence constraints, the predicates for bi-navigability must be adapted to the asssociation multiplicity in each case.

G RSL Specification - Examples

In this appendix the main parts of the RSL specifications generated from the class diagram shown in figure 1 are given.

G.1 RSL Specification for the example of domain class diagram (figure 1)

For each class present in the class diagram, two RSL modules are created. One has the specification for an object of the class, and the other, which uses the former, has the specification for the class. To illustrate, we present only the RSL specifications produced for the class "Product". The general structure of the specifications for the remaining classes is similar.

```
PRODUCT
object PRODUCTS =
  with TYPES in
  class
    type
      Product = PRODUCT.Product,
      Products = Product_Id →m Product

    value
      empty_Product : Products = [ ],

      add_Product :
        Product_Id × Product × Products ~→ Products
      add_Product(id, o, c) ≡ c † [id ↦ o]
      pre ∼ isin_Product(id, c),

      del_Product : Product_Id × Products ~→ Products
      del_Product(id, c) ≡ c \ {id}
      pre isin_Product(id, c),

      isin_Product : Product_Id × Products → Bool
      isin_Product(id, c) ≡ id ∈ dom c,

      get_Product : Product_Id × Products ~→ Product
      get_Product(id, c) ≡ c(id) pre isin_Product(id, c),

      update_Product :
        Product_Id × Product × Products ~→ Products
      update_Product(id, o, c) ≡ c † [id ↦ o]
      pre isin_Product(id, c),
```

```
            consistent : Products → Bool
            consistent(c) ≡
                (∀ id : Product_Id •
                    id ∈ dom c ⇒ PRODUCT.consistent(c(id)))
    end
```

TYPES

```
object PRODUCT :
    with TYPES in
    class
        type Product

        value
            description : Product → Text,
            price : Product → price,
            universalProductCode : Product → universalProductCode,

            update_description : Text × Product ~→ Product
            update_description(at, o) as o′ post
                description(o′) = at
            pre preupdate_description(at, o),

            preupdate_description : Text × Product → Bool,

            update_price : price × Product ~→ Product
            update_price(at, o) as o′ post price(o′) = at
            pre preupdate_price(at, o),

            preupdate_price : price × Product → Bool,

            update_universalProductCode :
                universalProductCode × Product ~→ Product
            update_universalProductCode(at, o) as o′ post
                universalProductCode(o′) = at
            pre preupdate_universalProductCode(at, o),

            preupdate_universalProductCode :
                universalProductCode × Product → Bool,
            describes : Product → Item_Id-set,
            contains : Product → ProductCatalog_Id,
            described_by : Product → SaleLineItem_Id-set,
```

update_describes : Item_Id-**set** × Product $\xrightarrow{\sim}$ Product
update_describes(a, o) **as** o′ **post** describes(o′) = a
pre preupdate_describes(a, o),

preupdate_describes : Item_Id-**set** × Product → **Bool**,

update_contains :
ProductCatalog_Id × Product $\xrightarrow{\sim}$ Product
update_contains(a, o) **as** o′ **post** contains(o′) = a
pre preupdate_contains(a, o),

preupdate_contains :
ProductCatalog_Id × Product → **Bool**,

update_described_by :
SaleLineItem_Id-**set** × Product $\xrightarrow{\sim}$ Product
update_described_by(a, o) **as** o′ **post**
described_by(o′) = a
pre preupdate_described_by(a, o),

preupdate_described_by :
SaleLineItem_Id-**set** × Product → **Bool**,
consistent : Product → **Bool**
end

The whole specification consists of a top level module "S" which uses the modules "SALES", "PRODUCTCATALOGS", "ITEMS", "PRODUCTS", "SALELINEITEMS", "PAYMENTS", "CUSTOMERS", "STORES", "POSTS", "MANAGERS", "CASHIERS" defined for each one of the classes present in the class diagram. Each one of these modules has the RSL specification for the corresponding class. The type "Sys" defined in "S" denotes the set of all the possible configurations for the system (in this case for the modeled domain). Each destructor in "Sys" returns a class container. Also a set of functions that operates on the class containers are defined in order to produce changes on the domain state. A particular function –named "consistent"– is defined to express all the desirable properties of the model, which is used, in turn, to write a set of axioms where consistency must be provided before and after any state changing function is applied.

SALES, PRODUCTCATALOGS, ITEMS, PRODUCTS, SALELINEITEMS,
PAYMENTS, CUSTOMERS, STORES, POSTS, MANAGERS, CASHIERS

object S :
with TYPES **in**

```
class
  type
    Sys ::
      sales : SALES.Sales ↔ replace_Sales
      productcatalogs :
        PRODUCTCATALOGS.ProductCatalogs ↔
        replace_ProductCatalogs
      items : ITEMS.Items ↔ replace_Items
      products : PRODUCTS.Products ↔ replace_Products
      salelineitems :
        SALELINEITEMS.SaleLineItems ↔
        replace_SaleLineItems
      payments : PAYMENTS.Payments ↔ replace_Payments
      customers :
        CUSTOMERS.Customers ↔ replace_Customers
      stores : STORES.Stores ↔ replace_Stores
      posts : POSTS.POSTs ↔ replace_POSTs
      managers : MANAGERS.Managers ↔ replace_Managers
      cashiers : CASHIERS.Cashiers ↔ replace_Cashiers

  value
    update_Sales : SALES.Sales × Sys ~→ Sys
    update_Sales(c, s) ≡ replace_Sales(c, s)
    pre preupdate_Sales(c, s),

    preupdate_Sales : SALES.Sales × Sys → Bool,

          . . .

    /* Here the remaining functions defined for
       each class container */

          . . .

    consistent : Sys → Bool
    consistent(s) ≡
      SALES.consistent(sales(s)) ∧
      PRODUCTCATALOGS.consistent(productcatalogs(s)) ∧
      ITEMS.consistent(items(s)) ∧
      PRODUCTS.consistent(products(s)) ∧
      SALELINEITEMS.consistent(salelineitems(s)) ∧
      PAYMENTS.consistent(payments(s)) ∧
      CUSTOMERS.consistent(customers(s)) ∧
      STORES.consistent(stores(s)) ∧
      POSTS.consistent(posts(s)) ∧
```

```
        MANAGERS.consistent(managers(s)) ∧
        CASHIERS.consistent(cashiers(s)) ∧
        (∀ id1 : Cashier_Id, id2 : POST_Id •
            (id1 ∈ dom cashiers(s) ∧
              id2 = CASHIER.records_sales_on(cashiers(s)(id1))) ⇒
               (id2 ∈ posts(s) ∧
                 id1 = POST.records_sales_on(posts(s)(id2)))) ∧
        (∀ id1 : POST_Id, id2 : Cashier_Id •
            (id1 ∈ dom posts(s) ∧
              id2 = POST.records_sales_on(posts(s)(id1))) ⇒
               (id2 ∈ cashiers(s) ∧
                 id1 =
                   CASHIER.records_sales_on(cashiers(s)(id2)))) ∧
         . . .

         /* Here the remaining predicates for checking
              binavigabily on the associations */

  axiom
    ∀ s : Sys, c : SALES.Sales •
        update_Sales(c, s) as s′ post consistent(s′)
        pre consistent(s) ∧ preupdate_Sales(c, s)

   . . .

  /* Here the axioms for the other defined functions */

   . . .

end
```

References

[1] http://www.omg.org.

[2] http://www.w3c.org.

[3] G. Booch. *Object-Oriented Design with Applications.* Benjamin/Cummings, Redwood City, CA, 1991.

[4] G. Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin/Cummings, Redwood City, CA, 2nd edition, 1994.

[5] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.

[6] P. Chen. The entity-relationship model - towards a unified view of data. *ACM Transactions on Database Systems 1*, 1976.

[7] S. DeLoach and T. Hartrum. A theory-based representation for object-oriented domain models. *IEEE Transactions on Software Engineering*, 26(6):500–517, June 2000.

[8] T. DeMarco. *Structured Analysis and System Specification.* Prentice-Hall, 1979.

[9] R. France. A Problem-Oriented Analysis of Basic UML Static Requirements Modeling Concepts. In *Proceedings of OOPSLA '99, Denver, CO, USA*, Nov 1999.

[10] C. George et al. *The RAISE Specification Language.* Prentice-Hall International (UK) Limited, 1992.

[11] C. George et al. *The RAISE Development Method.* Prentice-Hall International (UK) Limited, 1995.

[12] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming 8*, pages 231–274, 1987.

[13] Steve Holzner. *Inside XML.* New Riders, 2001.

[14] M. Jackson. *System Development.* Prentice-Hall, 1982.

[15] I. Jacobson et al. *Object Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley,Reading, Ma, 1992.

[16] Soon-Kyeong Kim and David Carrington. A Formal Specification Mapping between UML Models and Object-Z Specifications. In *LNCS*, number 1878, pages 2–21. Springer-Verlag, 2000.

[17] Zhiming Liu. Object Oriented Software Development using UML. Technical Report 229, UNU/IIST, March 2001.

[18] Zhiming Liu, Jifeng He, and Xiaoshan Li. Formalizing the Use of UML in Requirement Analysis. Technical Report 228, UNU/IIST, March 2001.

[19] H. Maruyama, K. Tamura, and N. Uramoto. *XML and Java, Developing Web Applications.* Addison-Wesley, 2000.

[20] E. Meyer and J. Souquieres. A Systematic Approach to Transform OMT Diagrams to a B Specification. In *Proceedings of FM '99*, volume I of *LNCS*, pages 875–895, 1999.

[21] E.R. Olderog and A.P. Ravn. Documenting Design Refinement. In *Procceedings of FMSP'00, Portland, Oregon*, pages 89–100, 2000.

[22] OMG. *OMG-Unified Modeling Language Specification v1.4.* http://www.omg.org/technology/documents/formal/uml.htm, September 2001.

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

[24] J. Rumbaugh, G. Booch, and I. Jacobson. *The Unified Software Development Process.* Object Technology Series. Addison Wesley, 1999.

[25] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Object Technology Series. Addison Wesley, 1999.

[26] A. Sutcliffe. *Jackson System Development.* Prentice-Hall, 1988.

[27] J. Warmer and A. Kleppe. *The Object Constraint Language.* Object Technology Series. Addison Wesley, 1999.