Acknowledgements

I would like to thank my advisors Jean Claude Royer and Annya Romanczuk for the great support devoted me during the development of this work.

I would like also to thank Pascal André, a collaborator of this work, whose ideas were really helpful in the composition of the semantic framework.

It was very important for me the long times of meeting we all had together.

I thank Prof. Dr. Cabral Lima and Universidade Estadual do Norte Fluminense for giving me the great opportunity of participating in this project.

The support given by my Brazilian friends during the entire Master was essential for me in order to conclude the works.

Finally, special thanks to my mother and my sisters that, even so far, gave me great incentive and love during the study period.

TABLE OF CONTENTS

List of Figures

List of Tables

Abstract

This Thesis describes an algebraic semantic framework covering the formalization of the main static model elements of UML. As UML is the unification of Object-Oriented analysis and design modeling languages, the formalization process presented here can be easily extended to other similar Object Oriented (OO for short) notation sets. Moreover it can contribute towards a standardization of OO modeling concepts. In the semantic framework presented in this work, model elements are formal described through algebraic specifications defining abstract data types. Abstract data types allow the specification of the semantics in an abstract way being really suitable to the description of OO models. The formal specifications are written in Larch Prover (LP for short) [GG91]. LP is a theorem prover that allows verifications and validations to be applied to the formal specifications. From these validations properties and inconsistencies about the models can be proved what leads to early detection of errors in the software development process. These formal specifications to be interpreted by LP are generated from a UML CASE tool built in Graphtalk metatool [CS97a]. The integration between the CASE tool and formal specifications is provided through a set of mapping rules established in this work.

Introduction

The goal of the thesis is to compose a semantic framework in order to support the formalization of the main static model elements of UML using Algebraic Specifications. Algebraic Specifications are used to describe abstract data types (ADT). The motivation for this work is the assumption that many Object Oriented (OO) methods, including those from which UML is derived, suffer from a lack of a precise *semantics*. This can lead to confusions and different interpretations when analyzing a model.

The semantic framework is based on a set of mapping rules defined to the translation from UML to algebraic specifications. These mapping rules are written in accordance to the syntax and semantics of each UML model element considered. The semantics of the model elements was evaluated considering the UML metamodel [UML99] and UML model [BRJ99a]. Therefore the result of the translation process is an ADT specified to each model element through the corresponding mapping rules established to it.

To establish these mapping rules defining the formal specifications some other approaches on Object Oriented analysis and design formalization, some of them focusing on UML, were evaluated and taken into account. In [LB98] a semantic framework for part of UML, named RAL, is presented. Another algebraic approach using Larch Shared Language (LSL for short) was also analyzed. It is described in [HHK98] being a formal, modular approach to specify the semantics of object-oriented models expressed in UML. LSL is an algebraic language, which in conjunction to LP and other technologies compose the Larch family of languages and tools

In both approaches a great importance is given to model theory composition in order to describe models and submodels. This allows the establishment of constraints among model elements. The level of granularity considered to the formal specifications is also an important aspect outlined in both approaches.

In the semantic framework presented here it is adopted an intermediate degree of granularity. Formal descriptions are used to describe classes and associations as well as some other constructs. It is also considered the idea of constraints at the model level what is achieved through general descriptions grouping some individual model elements.

The implementation of the semantic framework is undertaken considering the integration of different technologies: Graphtalk metatool, C++ programming language and Larch Prover theorem prover. Graphatlk metatool is instantiated with the UML grammar to build a CASE tool. The Graphtalk API primitives are used in the C++ source code allowing the automation of the translation process from a UML informal model to well-formed algebraic specifications. Larch Prover reads then these formal algebraic specifications in the form of abstract data types supporting that checks and proofs can be performed on them resulting in error detection on the design phase.

As the work of this dissertation considers just the static part of UML some side-effects on operations that depends on collaborations are not described. The formal method presented can be extended in future in order to cover also dynamic UML concepts.

Structure of the Dissertation

Chapter one presents an overview of the current state in UML semantics and gives some motivations and difficulties towards UML semantics formalization.

Chapter two shows the State of the Art in UML semantics formalization. It presents formal methods and formal languages that can be used to the formalization of Object Oriented analysis and design languages. In the core of the chapter is the presentation of three formalization approaches existent, two of them focusing specifically in UML, and the other one, Formal Classes approach, showing a more general formalization method that can be applied to any OO design and analysis language.

In chapter three the core of the Thesis is described. This chapter shows the main points considered to compose the semantic framework, as the formal syntax followed, the process to determine the mapping rules, the structure of an ADT, going then deep in the description of the set of mapping rules to each UML model element considered in the formalization. The mapping rules are described based on the semantics aspect that leads to their definition.

As explained in this introduction, the implementation of this semantic framework takes into account different technologies that need to be well integrated in order to allow the framework working. Each of these technologies employed and the way taken to their integration is explained in chapter four.

Chapter five gives then the link between the theoretical part presented in chapter 3 with the practical aspects detailed in chapter four. This chapter takes a UML Static Diagram drawn in the CASE tool developed as part of this work and shows the results of the translation process performed to it. Therefore the formal specifications in the form of ADTs resultant from the implementation of the mapping rules are referenced. After the translations are done, this chapter goes on presenting some inconsistencies that can be detected in UML models through the use of the semantic framework.

Chapter six ends up giving some conclusions taken during the development of this work and presenting contributions and future work that can be taken in order to complete the semantic framework and its practical application.

Chapter One

Motivations and Difficulties towards UML Formalization

UML, the Unified Modeling Language, is a very expressive language that can be used to model object-oriented software systems. It is the unification of the main object-oriented (OO) methods (Rumbaugh, Booch and Jacobson). The Object Management Group (OMG) has approved UML in November 1997 as the standard notation for object-oriented analysis and design.

The main motivation towards UML formalization is that its semantics is not precisely described through UML official documents and books. In this chapter other motivations and some difficulties encountered in order to achieve UML semantics formalization are presented.

1.1 UML Semantics: Current Form

UML encompasses structural and behavioral aspects in order to describe OO software systems. Even being a de facto standard, its semantics are semi-formal described. In [UML99], the *UML semantics* document, version 1.3 (last version), the semantics of the language is described using the metamodel. The metamodel stands a combination of graphical notation, natural language and formal language. It gives a syntactic description of the language but not a complete and precise specification of its semantics.

The graphic part is reflexive using a subset of the own UML notation. The formal language is the OCL (Object Constraint Language) that has been a first approach in order to get a precise description for the UML. It is an assertion language used to describe navigation and constraints in Class Diagrams (the static diagram of UML). Although OCL helps in the semantics description being used to the specification of well-formedness rules, it does not provide a basis for controls and validations. Moreover it does not solve some ambiguities in UML interpretations.

UML carries a complex set of notations that as explained do not gain a clear meaning through the metamodel.

1.2 Difficulties to UML Formalization

The lack of a precise formal semantics for the UML is justified in many ways:

- The architects of the language claim: *"the state of the practice in formal specifications does not yet address some of the more difficult language issues that UML introduces" [UML99].*

- ✍ Formal specifications are hard to deal with for non-expert users. Developers, users of UML, are not familiar with formal mathematical specifications and because of it they tend to resist to their use.
- ✍ To be of industrial use, formal specifications need to be integrated to CASE tools, supporting graphical modeling constructs, in such a way that developers can directly manipulate the OO models they have created to analyze, transform and enhance them.

In contrast to the difficulties showed above, the own authors of the modeling language also recognize the importance of formality. According to [CE97] the authors of the language agree in the sense that it lacks from a precise semantics description, and that its formalization could lead to unambiguous interpretations of the models and could permit extensibility allowing future changes in object-oriented analysis and design.

1.3 Motivations to UML Formalization

Many motivations are given to justify the importance of formalization. They can be grouped according to some primitives, as: clarity, consistency, correctness and enhancement it can bring to the models. Because of these benefits, formalization is really helpful in forward and reverse reengineering efforts as well as in the restructuring of systems. On the other hand, a really understandable and consistent system is more suitable for reuse. Follows some motivations towards formalization according to the primitives stated.

✍ **Clarity:**
UML is a complex language that holds a really great number of modeling elements. Because of its complexity and lack of precise description, its constructs are not clear defined and the language can lead users to ambiguous interpretations of the models. Formalization can help in clarifying the meaning of UML model elements. In [CE97] it is stated:
"Clarity acts as a reference – if at any point, there is confusion over the exact meaning of a particular UML component, reference can be made to the formal description to verify its semantics."
A deeper understanding of OO concepts is also gained, allowing the development of more rigorous semantic analysis tools and better use of OO techniques.

✍ **Consistency:**
UML presents nine different diagrams to express different system perspectives. The consistency among these diagrams representing a model can be ensured since all of them are formalized and hence precisely described. This leads to a more complete and unambiguous interpretation of a model, allowing development teams to have a better communication and understanding among them.
Consistency can also be achieved between code and specifications. Having a precise description of the models, implementations can be validated against the design checking if it fulfills the specifications. On the other hand, formalization can also be a bridge from implementation to design in a reverse engineering process.

✍ **Correctness:**
Correctness of the models can be achieved through the application of proofs over the formal specifications. Therefore inconsistencies can be detected. A mapping between the model elements of UML to formal specifications can help in adapting proofs and validations to CASE tools what leads to early detection of errors in the systems.
The establishment of proofs can be done upon the properties of a system described in UML, forming a basis for future automatic proof techniques.
Moreover with a mapping allowing the generation of formal specifications from informal models it is possible to identify ambiguous and inconsistent structures in the models.

✍ **Enhancement:**
Enhancement of models is expressed through design refinements. In [EBFLR98] refinement is defined as:
"It is the process by which an abstract model of a system (containing relatively little implementation detail) can be incrementally transformed into a model that can be readily implemented in a specific programming language. At each stage the correctness of the more detailed model must be verified against the abstract model."

As UML is a diagrammatical modeling language, refinement of a UML model implies a process of diagrammatical transformations. In this context, the definition of a set of semantically-based transformation rules is important to provide a set of correct transformations that are equivalencies or enhancements of models. Some properties of models can be deduced and proved through transformations. Proving that one form of the model is equivalent to another can make correct properties arise.
Refinements of models based on transformations are useful not only to support forward engineering as well as reengineering efforts. Model refinements can be helpful in the restructuring of designs.
Design Patterns can be applied in refinement steps being checked for correctness. Once checked, a pattern can be used again and again without having to be re-checked.

Basing in the primitives previously stated and going into detailed explanations, more justifications for formalization can arise. In [FELR97] they say:

- ✍ Developers can waste time making considerations over correct usage and interpretation of notations. Because of the informal descriptions provided in reference books, it is not easy to achieve an interpretation that can be considered precise.

- ✍ It is difficult to ensure model reviews, rigorous semantic analysis based on informal techniques. In [FELR97] it is stated:

"Review meetings can be further enhanced if the notations used have a precise semantics. The results of model validations and verifications can be presented in reviews as evidence of the quality of the models. Rigorous semantic analysis techniques also facilitate the early detection of modeling errors which considerably reduces the cost of error removal."

- ✍ Tool support for OO modeling notations is limited because of the lack of a precise semantics for the constructions of the language. Hence tools stay limited to cover just syntactic concerns.

In [EBFLR98] it is stated that:

"The desire to formalize UML was originally motivated by the overall wish to develop practical, industrial strength, formal methods. The advent of the UML as a likely de-facto industry standard, and its recognition that as a standard it needs to be precisely described, made UML a natural choice for a combined investigation."

As it can be realized the motivation to formalize OO methods was not originally motivated by UML emergence. Formalization had already been recognized as useful and necessary not only for academic purposes but also for industrial use before UML has appeared. Formalization aims to support reliable and precise modeling language to be used in any context. The advent of UML as a standard OO modeling language made the efforts turned to it.

Chapter Two

UML Semantics Formalization

In the previous chapter many motivations were given to justify the efforts invested in UML formalization. This chapter starts showing some formal methods and languages to support formalization. Afterwards, the main OO analysis and design formalization approaches studied, some of them focusing on UML, are presented.

2.1 UML Precise Group: An Important Work Taken Towards Formalization

Before presenting the formal methods and formalization approaches, it is necessary to point out the importance and contributions of the UML Precise Group in the context of UML formalization.

The authors of [EBFLR98] compound the UML Precise Group (PUML) which was created for two main purposes: investigate the completeness of the UML semantics and develop novel approaches to use UML more precisely. This group was formed in late 1997. By giving precise semantics to UML, the group intends to develop a formal *reference manual* for the language. In [FELR97] they say:

"A major objective of the project is to develop a formal reference manual for the UML. This will give a precise description of core components of the language and provide inference rules for analyzing their properties. In developing the reference manual we will build upon the semantics given in the UML semantics document by using formal techniques to explore the described semantic base."

In this formal reference manual, the intention is to re-express the formal semantics in terms of a suitably expressive language, that could be a mixture of notations such as an enhanced version of the UML metamodel, the OCL (Object Constraint Language), and precise natural language statements.

2.2 OO Formalization Methods Classification

The classification presented in this section is also a contribution work from some members of the UML Precise Group. In [FELR97] it is presented three general categories for OO formalization methods: *supplemental, OO-extended formal language, and methods integration.*

In the *supplemental* method, formal statements substitute annotations in the models that are expressed in natural language. This clarifies the meaning of the models, but the semantics of graphical constructs are not necessarily precisely defined.

In the *OO-extended formal language* method, an existing formal notation is extended with OO features. This is the case of Z++ and VDM++, for example. In this case the formal languages are really enriched and, on the other hand, OO concepts need to be formalized in order to be able to be adapted to formal languages. The problem with this method is the considerable gap between model elements representing real world concepts and the mathematical representations in the formal notations.

Methods Integration approach defines the generation of formal specifications from informal OO models. It is stated:

"…the generation of formal specifications from informal models is only possible if there is a mapping from syntactic structures in the informal modeling domain to artifacts in the formally defined semantic domain."

In this case a formal description of the mapping rules becomes essential in order to check if the formal specifications indeed capture the intended interpretations of the informal models.

2.3 Formal Languages Classification

In [CHS+97] four major underlying models upon which the formal specification languages can be based are described. Follows the identification of these models and examples of formal languages classified in each one of them.

✍ **First-order logic and set-theory.**
According to [CHS+97], this approach can be defined as:
"The first-order logic and set-theory approaches are also often called model oriented because they support the specification of a system by constructing a mathematical model for it."
In this group there are:
✍ Z language;
✍ Object-Z (OO extension of the Z notation);
✍ VDM++ (OO extension of the Vienna Development Method);
✍ Z++ (OO extension of the Z notation).

✍ **Algebraic approach.**
This approach uses algebraic equations in order to establish the semantics of the operations in a specification.
Examples of languages are:
✍ TROLL;
✍ Maude;
✍ AS-IS (Algebraic Specification with Implicit State);
✍ Larch;

✍ **Petri nets/algebraic nets.**
This approach is described in [CHS+97] in the following way:
"Petri nets and high-level nets are two representative of the model-based class in the sense that they describe the state of a system by means of places which contain "black tokens" for the conventional Petri nets and structured tokens for high-level nets. A set of transitions which consist of a pre- and a post-condition, describes how the system state changes by consuming and producing tokens in the various places of the net."

Examples of languages in this family are:
✍ CLOWN (Class Orientation with Nets);
✍ CO (Cooperative Objects);
✍ OPN (Object Petri Nets);
✍ COOPN/2 (Concurrent Object-Oriented Petri Nets).

✍ **Temporal logic.**
In [CHS+97] it is described as:
"Temporal logics are axiomatic formalisms that are well suited for describing concurrent and reactive systems. A common aspect associated with temporal logics is a notion of time and state."
Examples of languages are:
✍ TRIO+;
✍ OO-LTL.

Follows the description of two UML formalization approaches that deal with set-theory (Z) and algebraic formal languages.

2.4 Definitions in the Context of Formal Languages

Some definitions become necessary in order to understand the following OO analysis and design formalization approaches and the remaining of the document. They are:

What is an Abstract Data Type (ADT)

Originally data types are defined as sets equipped with operations. Considering Abstract Data Types many definitions can arise:

1. A class of data objects with a defined set of properties and a set of operations that process the data objects while maintaining the properties.
2. A set of values and a set of operations on those values.
3. In [Royer99a] an ADT (Abstract Data Type) is defined as:

"An Abstract Data Type is the description of a data type. This description is said abstract because the semantics are expressed as relations between operations."

What are Terms?

By terms it can be understood an expression that refers to an object as: sizeof(Array).

What is first-order logic?

By first-order logic it is understood that equations can be written using variables that represent all the values that can be extracted from a specific Universe. The equation can then be proved valid by exemplification.

2.5 Object Oriented Analysis and Design Formalization Approaches

2.5.1 Set-Theory Methods: RAL – Real-Time Action Logic

In [LB98] a semantic framework for part of UML is presented. The formal framework is termed Real-Time Action Logic (RAL). This name comes from the fact that it intends to reason about real-time specifications. The mathematical semantic representation of UML models is given in terms of *theories*. This is a Z-based approach.

A RAL theory has the form:
theory Name
types local type symbols
attributes time-varying data, representing instance or class variables
actions actions which may affect the data, such as operations, statechart transitions and methods
axioms logical properties and constraints between the theory elements

Theories can be defined to a whole model, submodels, or specific elements such as classes, associations, states, etc, being in this case assembled through theory *morphisms.*

The Z Language employed is presented in section 2.3.

✍ **Theory at the Model Level**
A theory for a model in this approach can be defined as depicted in figure 1 – example 1.

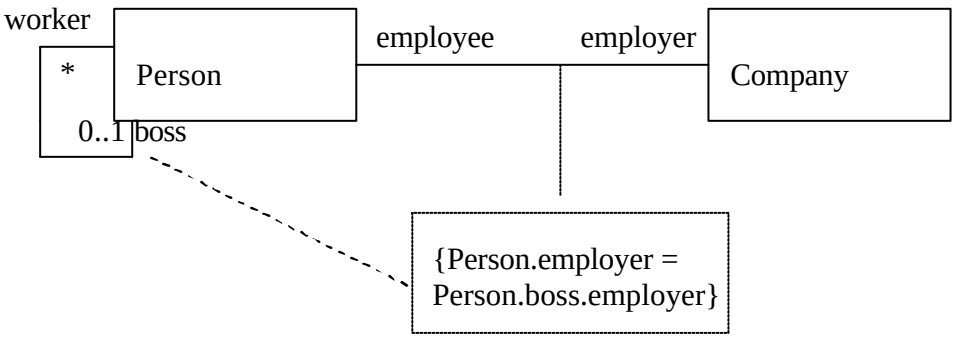

Figure 1. UML static diagram example

Example 1:

theory Employment
types Person, Company
attributes
Person: FIN(Person)
Company: FIN(Company)
employee_employer: Person ? Company
employee: Company ? FIN(Person)
employer: Person ? FIN(Company)
worker_boss: Person ? Person
worker: Person ? FIN(Person)
boss: Person ? FIN(Person)
actions Standard predefined actions to modify classes and associations:
create_Person(p:Person) {Person}
kill_Person(p:Person) {Person}
create_Company(c:Company) {Company}
kill_Company(c:Company) {Company}
add_link_worker_boss(p:Person, q:Person) {worker_boss, worker, boss}
delete_link_worker_boss(p:Person, q:Person) {worker_boss,worker, boss}

axioms
Constraints on the association links employee_employer:
forall p:Person; c:Company.(c:employer(p) ? (p,c):employee_employer
& p:employee(c) ? (p,c): employee_employer)
Cardinality Constraints:
forall p:Person.(card(employer(p)) <= 1)
forall p:Person.(card(boss(p)) <= 1)

The Constraint of the model is expressed by the formula:
forall p:Person.(employer(p) = employer[boss(p)])

In this theory, Person represents the finite set of existing objects of class Person. In the same way Company represents the set of Companies. Through the role employee in the Association between Person and Company it is possible to recover the set of existing objects of class Person linked to a Company. The same happens to the other association roles.

The actions determine the creation and deletion of objects, as well as the addition and deletion of links in associations.

✍ Representing a UML Class

A UML class is semantically represented by a theory T(C) of the form:

theory T(C)
types C
attributes
C: FIN(C)
self: C ? C
att1: C ? T1
…..
actions
create_C(c:C) {C}
kill_C(c:C) {C}
op_l(c:C, x:X1):Y1
…..
axioms
forall c:C.(self(c) = c & [create_C(c)](c:C) & [kill_C(c)]not(c:C))

Important points stated about this Class theory are:

1. Instance variables are modeled as attributes through a function type C ? T.
2. The notation [action]P denotes that every execution of action terminates with the predicate P being true. Thus create_C(c) always adds c to the set of existing C objects, and kill_C(c) removes it.
3. Class attributes and actions do not gain the additional C parameter as they are independent of any particular instance.

✍ Representing a UML Association

Associations are described through theories, which, as in the class theory, have an attribute representing the set of all links of the association. Therefore association theory also encompasses add_link and remove_link actions. Axioms determine the multiplicity of the association ends and other properties of the association.

✍ Representing Generalization (Inheritance)

Generalization is achieved through theory morphism. In [LB98] it is stated that:

"Generalization of class C by class D in UML is directly represented by the theory T(D) of D being the source of a signature morphism into T(C) which is the identity (each symbol of T(D) is interpreted by itself in T(C))."

"A theory morphism is a signature morphism s from T1 to T2 which preserves all the axioms of the source theory. That is, T2 proves s(P) for each axiom P of T1."

Theory morphism can be achieved by the inclusion of one theory (all its symbols and axioms) in another. Supposing we have a theory for a superclass T(C) and a theory for a

subclass T(D), adjoining the axioms can make the attributes and operations of C applicable to instances of D.

✍ **Defining Models by Composition**

Includes clause can also be used to another purpose. It can be used to compose models or submodels by assembling element model theories as depicted in the following example.

Example 2:
theory Employment
includes WorkerBoss, EmployeeEmployer
axioms forall p: Person.(employer(p) = employer[boss(p)])

The theory Employment showed in example 1 can then be rewritten in a simpler form just by including the theories of the associations, which in turn include the theories of the classes Person and Company. Therefore it is possible to realize that theories can be constructed by composition.

Composition is important to allow reuse. Theories defined in a high granularity level that can be assembled to define a model are more suitable for reuse.

2.5.2 Modular Algebraic Semantics for Object Oriented Models

In [HHK98] they define a formal, modular approach to describe the semantics of object-oriented models expressed in UML. The main aspect in this approach is to treat each individual model element as an entity that can be expressed through a theory (or trait) in Larch. The semantics of the model is then the composition of the semantic entities representing the individual model elements. It is stated:

"…this leads to a high degree of elegance and transparency in the semantics, any results proved about a generic trait or combination of traits will carry forward to models whose semantics has been built using them."

A high granularity to the formalization is considered as can be seen through the following list.

Elements list:
- ✍ Object-type (for class);
- ✍ Set of objects of the type;
- ✍ Association;
- ✍ Cardinality of the associations;
- ✍ Subtype (dynamic and static);
- ✍ Inherited attributes;
- ✍ Invariant;

- Diagram.

The motivation to this highly modular approach is that formalizing each element separated can increase reuse. Moreover formal descriptions can be used to specify components. Precision is really important in the specification of components and component interfaces, mainly when they are viewed as "black boxes" with hidden design and implementation. A user of a component needs a precise certificate about what the component does. They say:

"There is a natural progression from using this approach to build semantics of individual models, to use it to compose models into larger models. This is what is required to support component-based development, where components are specialized and composed to build other components and, eventually, systems."

In this approach it is used the Larch Shared Language (LSL for short), in which specification modules are called traits. Traits are used to describe abstract data types and theories having the following structure:

SpecName(parameters): **trait**
includes
existing specification modules to be used
introduces
function signatures are listed here
asserts
axioms are listed here

✍ Representing a UML Class

In this approach classes are referenced as object types, and class diagrams as type diagrams. The basic function in an object type specification is the one that can return the set of existing objects of that type at a point in time. Considering an object of type **A**, this basic function will have the following signature:

A **:** ? **-> Set[A]**

In which, ? is the sort representing system states and **Set[A]** is the sort of finite sets of elements of sort **A**. Therefore A(?) expresses the finite set of existing **A** objects in the state ?.

Instance attributes for object types are represented as functions mapping the attribute name to a value from a given type as follows.

attr1 : A, ? **-> T1**

Therefore to specify an object type it can be defined a basic trait including the function that allows manipulation of the set of existing objects (example 3), and a trait including

functions for the object type attributes (example 4). This strong separation of concerns in trait specifications due to the high modularity desired.

The final trait for the object type is then constructed by including the other two traits as shown in example 5.

```
Basic-Object-Type (A, A): trait
        includes
                Set(A)
        introduces
                A : ?  -> Set[A]
```

Example 3: Basic Object-Type Trait

This trait specifies object types by renaming **A** and A**.**

```
Attributes-Object-Type (A): trait
        includes
                T1, T2
        introduces
                attr1 : A, ? -> T1
                attr2 : A, ? -> T2
```

Example 4: Attributes Trait

```
Object-Type A: trait
        includes
                Basic-Object-Type (A, A), Attributes-Object-Type (A)
```

Example 5: Object-Type trait

✍ **Representing a UML Association**

Given the classes **A** and **B** associated in a UML class diagram, the plain association between them could be represented through two mapping functions with the signatures:

a : Set[instancesB], ? **-> Set[instancesA]**
b : Set[instancesA], ? **-> Set[instancesB]**

In which **a** and **b** represent the role names that maps a set of objects of one type to a set of objects of the other type. Through these mapping rules, associations are uniformly described. These functions can also be expressed through the signatures:

a : B, ? **-> Set[A]**
b : A, ? **-> Set[B]**

Mapping just one object to the connected set of objects of the other type.

✍ Specifying Multiplicity Constraints

Taking the previous association between **A** and **B**, an axiom to determine a one-to-many multiplicity has the form:

a: A **(**:**)**: **size(b(a,** :**)) = 1**

Where size is a set operation that returns the number of objects in a set. This axiom constraints the multiplicity in **B** by determining the size of the **Set** of **B** elements associated to an **A** element equal 1 (see also the previous functions in Representing a UML Association). Size operation upon sets is used to determine all the possibilities of multiplicity constraint.

✍ Subtyping

Subtyping is defined as:

"Subtyping is a special relationship between two object types, known as the is-a relationship."

In subtyping the subtype must be used anywhere the supertype is applicable and it inherits all the attributes and associations of the supertype. Considering the following example of inheritance between **A** and **B**, functions to express subtyping are given.

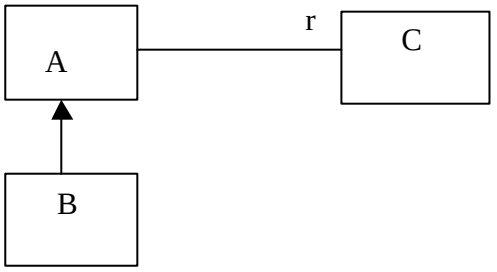

Figure 2. Subtyping

Two functions are used to express subtyping: ***simulates*** that maps an object identifier of type **B** to the corresponding object identifier in **A** that behaves like it, and ***member*** that tests if a **B** can be viewed as an **A**. They are expressed in the following way:

simulates: B -> A
memberB : A, : **-> Bool**

✍ Inherited Attributes

Simulates function is used to the description of inherited attributes. Considering an attribute **f** in class **A**, the following axiom to simulate it is also an attribute in **B** is written:

assert
f(b, : **) == f(simulates(b),** : **)**

✍ **Inherited Associations**

On the other hand, inherited associations are represented using role names. Considering the example in figure 2, the association between **A** and **C** with role name **r** is translated through the following function:

r : Set[A], ? **-> Set[C]**

To represent the inherited association, a new function is introduced:
r : Set[B], ? **-> Set[C]**

This new function **r** is constrained by the following axiom:
r ({b}, ? **) == r({simulates(b)},** ? **)**

✍ **Defining Models by Composition**

A specification for a type diagram is constructed just by including all the traits defined such as: object types, associations, cardinality constraints, subtypes, inherited role traits and invariant traits if defined.

2.5.3 The Formal Class Approach

In [RAC94] it is defined an algebraic approach to describe Object-Oriented analysis and design models in a formal way. The motivation to formalize Object-Oriented analysis and design is done through:

"Object-Oriented analysis and design need formal specifications to allow proofs, verifications and automatic processing."

The idea is to use the notion of formal class to build the formal specifications. A formal class is an abstraction of concrete class in languages like C++, Eiffel, CLOS or Smalltalk. It is an algebraic specification (as abstract data type) with an object orientation. The motivation to use an algebraic specification as abstract data type comes from the following assumption:

"Object-Oriented Design is the construction of software systems as structured collections of abstract data type implementations." [MEY88]

The specification model corresponds to modular design where formal classes are modules. The main concepts on a formal class structure are described as follows.

✍ Class Description

It is considered that a class defines a type and therefore inheritance implies subtyping. A class defines an aspect, which allows to abstractly describe its instances. It also defines the instance behavior.

The aspect has two parts: an abstract structure composed by a set of field selectors and a constraint that is a predicate on this abstract structure. A field selector has the profile: $\mathbf{fsel}_i$: **CFC -> $\mathbf{T}_i$**. In which, CFC represents the current formal class.

✍ Method Classification

A method is characterized by a profile, a precondition and axioms. Two main groups of operations (resp. methods) are distinguished: ***constructors***, the ones that have the current class as resulting type, and ***observers***, the ones having another type as resulting type. Field selectors as previously described are ***observer*** methods.

Another classification given to methods is accordingly to the user point of view. In this classification there are ***primitive methods***, the ones associated to the class aspect, and ***secondary methods***, the ones which semantics are based on the primitive methods. The basic constructor of a class (**new<CFC>**) is a primitive method. Field selectors are ***primitive observers***.

Secondary methods can be viewed as functional extensions of primitive ones. Their semantics is based on primitive methods, i.e. every application of a secondary method can be reduced to applications of primitive ones.

✍ Inheritance

The principles for inheritance in formal class model are:

- ✍ only secondary methods are inherited;
- ✍ redefining and masquing a method is possible;
- ✍ an inheritance link is possible between two classes if the following criterion is true: every field selector of the superclass exists in the subclass with the same type or a subtype.

According to the last principle, it can be realized that there is no inheritance of instance variables.

✍ Type Checking

The type checking is based on usual principles:

- ✍ there are predefined types as Boolean, Integer, String and generic List[T];

- a class defines a type;
- variables and methods are typed;
- inheritance implies subtyping;
- typing a message is like typing an operation application;
- the method to apply is selected on the receiver type;
- methods are redefined according to a rule which is co-variant only on the receiver type and the resulting type.

In this context it is possible to define a type checking and prove the safety of the control.

- **Rewrite Rules**

An abstract operational semantic to the model is given using conditional term rewriting, i.e. one operation is rewritten using another. This is valid for secondary observers that are expressed through primitive ones, and also for primitive observers that can be rewritten based on constructors. For field selector, for example, the following rewritten form can be obtained:

$$\mathbf{fsel_I(new<CFC>(X_i, \ldots, X_n)) \rightarrow X_i)}$$

- **Implementation**

Translation from formal classes to OO programming languages is quite natural and partially automatic. Such process takes as input the formal description and produces the "skeleton" of the class: class interface, class implementation, class structure, primitive methods code and secondary methods signature.

2.6 Conclusion and Summary

From the approaches presented in this chapter some meaningful ideas that can be reused in algebraic specifications defining ADTs are taken into account in the work of this thesis.

From RAL approach it is mainly considered:

- The representation of class attributes (and class operations) through a function that does not need an instance of the corresponding type as a parameter.
- The importance in adjoining theories to specify a more general sort (or theory) in order to be able to establish constraints among model elements.

About generalization, the formal definition in RAL is maybe not enough to express the needed semantics. Moreover it is not so clear how the axioms are adjoined in order to

allow that operations and attributes of the general type be applicable in the specialized one.

From the he study of the Modular Semantics approach, as it also makes use of an algebraic language, lots of benefits are taken. They will be realized through the description of the semantic framework in chapter 3.

As the intention is to allow rapid prototyping (refinement of models into code), proofs and verifications to be applied to OO analysis and design, much Formal Class principles are reused in the algebraic semantic framework proposed in this dissertation.

2.6.1 Comparison Table

The two approaches described for UML formalization are in fact the most complete, clear and concrete encountered. To provide a clear view of what each approach covers or not considering UML static aspects including model formalization, a comparison table is presented.

UML elements/ Formal Approaches	***Z based RAL***	***Algebraic Modular Semantics***
Class	✍ a theory ✍ represents the set of all existing instances through an attribute	✍ an object type trait including basic object-type and instance attributes traits ✍ considers the set of existing instances through a mapping function
Association	✍ a theory ✍ represents the set of all existing links	✍ a trait ✍ constrained by cardinality traits ✍ defines mappings between role names and sets of object types
Composition	No representation	No representation.
Generalization	✍ Achieved through theory morphism, i.e. the inclusion and mapping of operations and axioms of one theory into another.	✍ A function simulates is defined to map object identifier of the subtype to object identifier of the supertype. ✍ Simulates: B -> A ✍ This function allows attributes and associations of supertypes to be also applicable to instances of

		subtypes.
Instance Attributes	✍ Explicit in the class theory ✍ Attr1: C -> T1	✍ a trait is defined to specify the instance attributes ✍ attr1: object-type, ? -> T1
Class Attributes	✍ explicit in the class theory ✍ attr1: -> T1	No representation.
Instance Methods	✍ explicit actions in the class theory ✍ op_l(c:C, x:X1):Y1	No representation.
Class Methods	✍ explicit actions in the class theory ✍ op_l(x:X1):Y1	No representation.
Abstract Classes	No representation	No representation.
Interfaces	No representation	No representation.
Constraints	✍ cardinality constraints ✍ constraints between model elements	✍ for association cardinalities ✍ for subtyping, as disjoint subtyping constraint ✍ invariants written in OCL are translated
Model	✍ theories assembled by theory morphisms ✍ a whole theory defined with all model elements	✍ model specification (or diagram specification)

Table 1 – Comparison over UML Formalization Approaches.

Model is considered in the table because it is really important to specify theories that allow manipulating model elements together.

Chapter Three

UML and ADT: A Semantic Framework Proposition

The semantic framework proposed in this work is based on algebraic specifications describing Abstract Data Types (ADT). In the previous chapters the importance of UML formalization and some approaches in this direction have been presented. From these approaches some important outlined points are taken into account. The goal of this chapter is to explain the algebraic formal semantic framework through the mapping rules that support the translation from UML model elements to algebraic ADTs.

3.1 Main Points Considered in the Framework Composition

In order to compose the formal framework, the semantics of the main UML static model elements was evaluated. The main motivation towards UML formalization is the fact that the semantics of the UML model elements is not precisely described in the official UML semantics document [UML99]. Consequently, in some ambiguous points it was necessary to have recourse to other sources of information to achieve a good interpretation. Long times of discussion were also necessary to achieve final conclusions.

According to the final interpretation of the semantics, the mapping rules were defined having as a result the algebraic formal specifications for some UML static constructs. This process follows the directives of *Methods Integration approach* formal method the one chosen as the basis to this work. This choice is justified in the next section.

To start with the formalization, in this work it is considered the UML core concepts respecting to the structural aspects of the UML, which are:

- *Types* - implemented through Classes;
- *Instances* - objects of a type;
- *Values* - a type defines the values of its instances and the value of an instance consists of the values of its attributes at a point in time;
- *Operations* – description of the services that objects of a class can offer to others affecting their behavior;
- *Associations* – reflects structural relationships between classes;
- *Hierarchy and Inheritance* – types from a hierarchy in which inheritance of structural (attributes) and behavioral features from super to sub-types take place.

As in [CE97], the core concepts are extracted from the Core Object Model specification presented by Houston and Josephs [HJ95] written in Z that captures a precise description of the Object Management Group's emerging standard for objects.

Starting from the core concepts it makes feasible that future extensions to the semantic framework can be easily proceeded.

Another important aspect to point out is that the semantic framework presented is typed. However it is assumed that once translations to algebraic ADTs are proceeded, type-checking problems are not carried to the specifications. The ADTs are written in Larch Prover as will be shown in section 3.1.3.

3.1.1 The Formalization Method Chosen

The approach chosen for the formalization is the integrated one, called *Methods Integration approach* (see section 2.2). This approach is justified in many ways:

- A mapping between graphical and formal constructs can uncover problems with the modeling notations;
- It can help identifying ambiguous and inconsistent structures;
- It can help defining semantically well-formed informal models;
- The mapping rules can be adapted to a CASE tool in such a way that formal specifications can be automatic generated from informal models (to express the whole or at least part of the models). This can help in proving properties of the models and in generating code from them.

The integration of the translation process to a CASE tool built in Graphtalk metatool is explained in chapter 4 with a concrete example of the translations given in chapter 5.

The mapping rules making the bridge from UML models to formal models are explained in section 3.2.

3.1.2 The Formal Language Chosen

The language used to write the formal specifications is Larch more specifically with the syntax of Larch Prover. It is an algebraic method not yet extended with OO concepts. However Larch is really suitable to the description of Abstract Data Types because it allows the semantics of the operations to be described in an abstract way, i.e. just as equations stating relations between them. In addition Larch Prover allows verifications and proofs to be applied to the formal specifications. This is really helpful in order to ensure the correctness of the models described. More information on Larch Prover is found in chapter 4, section 42.

3.1.3 ADT Structure

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

- A heading containing information about the module, mainly they are: the name (or sort) of the type, the imported modules (or types), and the generator names (or constructors).
- The signatures which describe the operators syntax.
- The axioms which describe the semantics of operations.

As in Formal Classes (section 2.5.3), primitive observers (operations related to the main aspect of the ADT) are described in terms of the constructors and secondary observers in terms of primitive ones. Constructors (or generators) are operations that are able to determine the values for the type being described. These assumptions are realized through the axioms in the following ADT.

The ADT example presented here specifies a sort Set, where ~ is logical not, /\ is and, \/ is or, => is implication and = is equality.

It is followed Larch Prover syntax. The reserved words of Larch are in Italics. Notes are between slashes.

set name **SetA** /defines the name of the sort – SetA – a set of A elements/
declare sorts **A, SetA, Nat** /declares the types used in this specification/
declare variables **a, a1: A, Xsa, Ysa: SetA** /declare the variables with the corresponding types that will be used in the axioms/
declare operators /defines the operators that apply to the values of the type being defined/

{}:	**-> SetA**	/operation that creates an empty set/
{_}: A	**-> SetA**	/receives an element and identifies the set in which it is present/
insert: A, SetA	**-> SetA**	/inserts an element in the Set/
_ \U _: SetA, SetA	**-> SetA**	/union of two sets/
_\in _: A, SetA	**-> Bool**	/tests if the element is in the set/
_\I _: SetA, SetA	**-> Bool**	/tests if one set is included in the other/
size : SetA	**-> Nat**	/returns the number of elements in SetA/

..
assert /semantics of the operations are described through the axioms written in the assert section/
sort SetA *generated by* {}, insert; /constructors of the sort SetA/

{a} = insert(a, {}); /a set with an a element is equal the insertion of a in an empty set/

~(a \in {}); /an a element is not in an empty set/
a \in insert(a1, Xsa) ? **(a \eq a1** ? **a \in Xsa);** /a in insert a1 in set Xsa is equivalent to that a is equal a1 or a is in Xsa/
{} \I Xsa; /empty set is included in a set/
insert(a, Xsa) \I Ysa ? **(a \in Ysa** ? **Xsa \I Ysa);** /insert an a element in set Xsa is in set Ysa is equivalent to a is in Ysa or Xsa is in Ysa/
a \in (Xsa \U Ysa) ? **(a \in Xsa** ? **a \in Ysa);** /an a element in Xsa set union to Ysa set is equivalent to a is in Xsa set or a is in Ysa set/
% axioms for size operator /comments begin with %/
size({}) = 0; /the number of elements in an empty set is 0/
(a \in Xsa) => size(insert(a, Xsa)) = size(Xsa); /the number of elements in set Xsa inserting an element that already existed is equal the number of elements originally in set Xsa/

~(a \in Xsa) => size(insert(a, Xsa)) = 1+size(Xsa); /if a is inserted in Xsa and didn't exist before, then the size of Xsa will be the original size + 1 element/

..

In the previous structure it is possible to see that the axioms are compound from equations that are equalities or equivalencies between terms with variables. Variables represent a valid value inside a Universe of its type.

These axioms are translated as rules in LP that are applied to the system any time it runs in LP to be tested. Through these rules the semantics of the system (composed by the semantics of each element) can be checked, properties validated and inconsistencies detected.

3.2 The Translation Process from UML to ADT

In order to make clear the translation process from UML static model elements (expressing the UML core concepts) to algebraic ADTs, the semantics of the model elements according to the *UML Semantics Document* [UML99] and to *The Unified Modeling Language User Guide* [BRJ99a] is presented. The semantics is presented focusing on the main points considered to the formalization in this work. Afterwards some considerations on the semantics according to the studies and discussions undertaken are described.

The translation process is also described in two parts: first the translations that result in the operations applicable to the type being defined are described (see declare operators section in the previous ADT structure), after that the most significant axioms determining the semantics of these formal operations are defined (see assert section in the previous ADT).

The result of the translation process is one ADT specified to each model element considering classes, associations (plain associations and compositions), generalizations, association classes and constraints for the moment. Some other ADTs of auxiliary types used in the formal specifications are also specified in the semantic framework, such as: primitive types (String, Nat, etc), identity for classes, and set of objects of a class. They are described in appendix A.

Follow the descriptions of the mapping rules for each model element considered in the framework.

3.2.1 Class Translation

✍ **Class Syntax and Semantics**

In the UML static diagram the main building block is the Class. A Class is the abstraction of a set of objects with the same properties (attributes), behavior (operations implemented through methods), relationships, and semantics. For the attributes, each object of a class has its own values, what characterizes particular concrete states for the objects. The values for the attributes are taken from the set of values permitted by the attribute type.

The behavior is shared by all the class instances. Class in UML is represented as showed in figure 3.

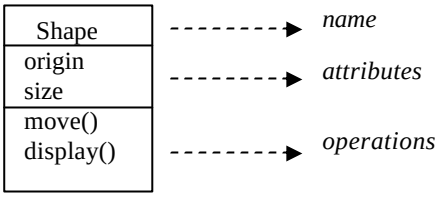

Figure 3. UML class representation

The ability to describe behavioral and structural (attributes) features are inherited from classifiers. Classifiers are defined as:

"A Classifier is an element that describes behavioral and structural features; it comes in several specific forms, including classes, data type, interface, component, and others." [UML99] (Classifiers - pg. 2-27)

The operations and attributes of a class have an important feature that is their owner scope. It can have two different values:

- Instance: each instance holds its own value for the feature (in case of attributes) or the feature is applicable to the set of instances of the class (instance methods);
- Classifier: there is just one value of the feature for all instances of the class (class attributes) or that the feature is applicable to the class itself (class methods).

According to [BRJ99a], examples of class attributes and operations can be:
"The most common use of classifier scoped features is for private attributes that must be shared among a set of instances, such as for generating unique Ids among all instances of a given classifier, and for operations that create instances of the class." (chapter 9 – pg. 124)

Classes are identified by their name. In [BRJ99a] it is stated that:
"Every class must have a name that distinguishes it from other classes." (chapter 4 – pg. 49)

It is also stated that:
"…the same thing in a system (such as the class Person) may appear multiple times in the same diagram or even in different diagrams. In each case, it is the same thing." (chapter 7 – pg. 94)

Besides these features stated to classes there are many others that apply. However to start with the formalization only the main features are considered as a basis. From the core description of classes, it is possible to extend the formal specifications in future to adapt what more becomes necessary.

✍ **Considerations on the Semantics**

A class can be viewed as the implementation of a type since it determines the operations applicable to a set of instance values. In fact classes can be viewed as implementations of different types through the realization of different interfaces (collections of operations determining the services of a class). However, for the purposes of this work classes and types are considered as semantically equivalents.

Instances of a class mean the objects of that class. In [CE97] instance is defined as the instantiation of a type with a unique identity. In the UML it is agreed that instances have unique identity.

✍ **Mapping Rules**

Considering a generic class A, to which general class formal specifications can be determined, the following set of mapping rules is established:

1. As in Formal Classes [Royer99a] (see section 2.5.3), a single generator (or constructor) is considered:
   newA : T1,…,Tn -> A

2. Primitive observers are defined for each argument type of the generator. They describe the instance attributes.
   getAttr1 : A -> T1
   setAttr1 : A, T1 -> A

3. Other instance operations are defined as functional extensions of these previous formal operations.

4. For object identity, two operations are defined:
   identity : A -> IdA
   __\eq__ : A, A -> Bool
   The identity operation that expresses the object identity, and the object equality operation that will be defined as identity equality (see rule 8).

   It is taken a functional model for object identity in which the identity is part of the values of the object. Therefore the constructor of the ADT gains this new signature:
   newA : IdA, T1, …,Tn -> A

   In which IdA represents the type for identity of objects of type A. IdA is defined as an ADT, which is described in appendix A.

5. Constants are defined in order to give examples of instances of the class that will be used later to test some axioms.
   oneA : -> A
   anotherA : -> A

Follows now the description of the axioms stating the semantics of these formal operations.

6. Primitive Observers, i.e. the instance attribute accessors, are described in terms of the constructor newA.
   getAttr1(newA(id1, var1, …, varN)) = var1;
   In which id1 is a variable expressing an identity value, var1 and varN are variables expressing values of the attributes according to their types. Then the operation getAttr1 obtains the value of the attribute represented by var1 in the constructor.

   setAttr1(newA(id1, var1, …, varN), var11) = newA(id1, var11, …, varN);

   Var11 represents another variable, i.e. another value for the same attribute expressed through var1. The constructor newA represents the parameter of type A in the set operation. Then the setAttr1 operation changes the value of the attribute represented by var1 for the object.
   These axioms are applicable to all the attribute accessors of the class.

7. Identity operator, expressing another primitive observer, is also described through the constructor.
   identity(newA(id1, var1 ,…, varN)) = id1;

8. As mentioned in item 4, object equality is achieved through identity equality.
   a1 \eq a2 = identity(a1) \eq identity(a2);
   In which a1 and a2 are variables of type A. The operation equality (\eq) between identities is defined in the ADT for the type IdA.

9. For the constants, the following axioms take place.
   oneA = newA((newIdA), var1, …, varN);
   anotherA = newA((nextIdA(newIdA)), var1, …, varN);

   The first constant oneA is equal a new instance of A with a new identity. And, anotherA represents an instantiation of A with a new identity obtained through the existing one. The newIdA and nextIdA are operations of the type IdA.

As LP does not allow genericity (see section 4.2), for each class a sort identity (as IdA for class A) is defined.

The explicit identity adopted does not correspond exactly to the implicit one considered in object-oriented programming languages, but it is not so simple to treat implicit identities in algebraic ADT. Implicit identities bring some side-effects and the way in which it can be solved is still under investigation by the collaborators of this work.

✍ **Metalevel Specifications**
In order to specify class attributes and class methods (according to the owner scope explained in class syntax and semantics), a specification for a class in a more global level can be given. This specification is considered at the metalevel since the own class is the

generated object in this case, and therefore the operations and attributes described are applied to the class.

Assuming the same generic class A, the following mapping rules are defined at the metalevel. It is important to state that the sort defined in this case is the sort **classA**.

10. For class creation:
    newClassA : String, T1, …, Tn -> classA
    In which String type is used to represent the class name (its identity). The newClassA is the constructor (or generator) of the ADT. T1,…,Tn represent the class attribute types.

11. According to the semantics, a name identifies a class. Then the following operators take place:
    classIdentity : classA -> String
    __\eq __ : classA, classA -> Bool
    In which class equality is achieved through identity equality.

12. As the class A must be the unique instance of the type, a constant to refer to it is defined. It will be used in the axioms instead of variables of the type.
    theClassA : -> classA

13. Class Attribute Descriptors:
    getClassattr1 : -> T1
    setClassAttr1 : T1 -> T1
    As the class attribute refers to the class itself and not to one of its instances, it does not gain an additional parameter of type A in its signature (as in RAL approach, section 2.5.1).

14. Class operations are defined as functional extensions of these previous operations.

Another point is that in the semantic framework the operation new is defined as the constructor for the sort, therefore the new for class instantiation is not considered at the metalevel as in Smalltalk approach even being considered a class operation in UML semantics.

3.2.2 Association Translation

✍ **Association Syntax and Semantics**

Associations are a structural relationship that can be established between classes. They define a set of tuples relating instances of the connected classes. Associations can include two or more association ends (the connection from the association to a class). In this work it is considered just binary associations for the moment as depicted in figure 4.

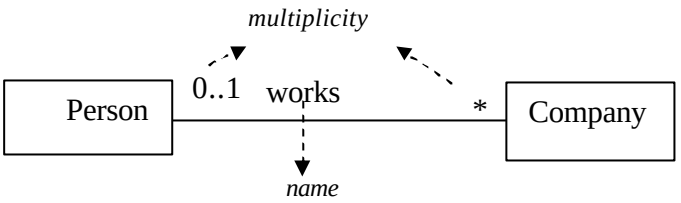

Figure 4. UML association between classes Person and Company

Associations can have a name, or it can be used role names to identify the association. Role names represent the role that the classes play in the relationship. Another important property of associations is the multiplicity. Each association end has its own multiplicity stating "how many" elements of that class can be related to an instance of the class in the opposite end. According to [BRJ99a], multiplicities can be: exactly one (1), zero or one (0..1), many (0..*), or one or more (1..*). An exact number or intervals are also possible.

An important restriction about associations, according to [UML99] is:
"The instances of an association are a set of tuples relating instances of the classifiers. Each tuple value may appear at most once." (Association – pg. 2-19)

Taking into account these points of association semantics, the following translation for association is done.

✍ **Mapping Rules**
It is important to state that in the semantic framework associations are identified by a natural number since association names are not always provided. Role names are not yet considered. They are really close to interface aspect: a role can have its type determined by an interface, i.e. a role an abstraction presents to another can be determined by the service it provides. So it makes more sense to include role names when interfaces are also treated in the framework.

Considering a generic association Assoc1 between classes A and B, the following set of mapping rules is established:

1. First of all, an empty association (as an empty set) is considered to which links can be added and removed.
   void : Nat -> Assoc1
   In which Nat represents the type for association identity, which needs to be treated in LP.

2. The other generator (or constructor) for the association sort is addLink:
   addLink : Assoc1, A, B -> Assoc1
   In which one instance of A and one instance of B are added as a tuple to the set of tuples represented by the association. The AddLink together with void operator determines the values for the association sort.

3. To allow the manipulation of the set of links, operation remove link is also described.
   removeLink : Assoc1, A, B -> Assoc1

4. AllLeftLink and allRightLink operators map an object of a given type to a set of related objects of the other type.
   allLeftLink : Assoc1, B -> SetA
   allRightLink : Assoc1, A -> SetB

5. Still considering links, an operator to test if two instances are related through the association is specified.
   isLinked : Assoc1, A, B -> Bool

6. Operators to test if an instance is linked through the association.
   isLeftLinked : Assoc1, B -> Bool
   isRightLinked : Assoc1, A -> Bool

7. Testing if the association, as it is a set, is empty.
   isEmpty : Assoc1 -> Bool

   The operations in rules 5 to 7 are necessary to write some proofs in the theorem algebraic prover LP.

8. Operators leftMultiplicity and rightMultiplicity are determined in order to express the left and right multiplicity of a given instance in the association. These operators could also be obtained by the size of the set of instances recovered through the operators allLeftLink and allRightLink (rule 4). However they are defined in order to get a more complete association description.
   leftMultiplicity : Assoc1, B -> Nat
   rightMultiplicity : Assoc1, A -> Nat

9. For association identity, the following operations are described.
   identity : Assoc1 -> Nat
   __\eq__ : Assoc1, Assoc1 -> Bool
   In which association equality is achieved through identity equality (see rule 12).

The main axioms determined in order to reflect association semantics are as follows.

10. According to association semantics, an axiom stating that tuples of instance values cannot be equal in an association becomes necessary.
    (a1 \eq a2) ∧ (b1 \eq b2) => addLink(addLink(assoc1, a1, b1), a2, b2) = addLink(assoc1, a1, b1);
    In which a1 and a2 are variables of type A, b1 and b2 variables of type B, and assoc1 a variable representing the association. Adding two links with variables that represent the same objects is like adding this link only once.

11. Multiplicity constraints are written only if the multiplicity is not free, i.e. different from 0 or More. Axioms for multiplicity constraint make use of the size operation for Set. Size returns the number of objects in a Set (as in the algebraic approach of section 2.5.2).
    Considering just the multiplicity at the right end the following axioms are described.
    %multiplicity Just One
    size(allRightLink(assoc1, a1)) = 1;

    %optional multiplicity (0..1)
    ~(size(allRightLink(assoc1, a1)) > 1);

    %minimum multiplicity 1, in the one or more (1..*) case
    ~(size(allRightLink(assoc1, a1)) < 1);

The axioms in items 12 to 17 state the semantics of some formal operations defined in the association specification.

12. Stating the semantics of the operation allLeftLink (that can also be applied to allRightLink with the adequate changes).

    (b1 \eq b2) => allLeftLink(addLink(assoc1, a1, b2), b1) = insert(a1, allLeftLink(assoc1, b1));
    What says that: if b1 is equal b2, then the result of allLeftLink to b1 adding a link to b2 will be the set resultant from allLeftLink to b1 plus one more element a.

    ~(b1 \eq b2) => allLeftLink(addLink(assoc1, a1, b2), b1) = allLeftLink(assoc1, b1);
    What says that : if b1 is not equal b2, then the result of allLeftLink to b1 adding a link to b2 will not afect the result of allLeftLink to b1.

    allLeftLink(void(i), b1) = {} :SetB ;
    The result of allLeftLink to an empty association is an empty set. The parameter i in the operation void represents association identity.

13. Axioms for association identity.
    identity(void(i)) = i;
    identity(addLink(assoc1, a1, b1)) = identity(assoc1);
    assoc1 \eq assoc2 = equal(identity(assoc1), identity(assoc2));

    In which equality between associations is obtained in LP through identity equality.

14. Axioms to state that an association is empty.
    isEmpty(void(i));
    ~(isEmpty(addLink(assoc1, a1, b1)));

15. Axioms stating when two instances of object types are linked

~(isLinked(void(i),a1, b1));
isLinked(addLink(assoc1, a1, b1), a2, b2) = ((a1 \eq a2 /\ b1 \eq b2) \/ isLinked(assoc1, a2, b2));

16. Axioms stating when one instance is linked through the association.
~(isLeftLinked(void(i), b1));
isLeftLinked(addLink(assoc1, a1, b1), b2) = ((b1 \eq b2) \/ isLeftLinked(assoc1, b2));
To the operator isRightLinked the same axioms are valid making the adequate changes.

17. Axioms stating the semantics for leftMultiplicity and rightMultiplicity operators (rule 8).
leftMultiplicity(void(i), b1) = 0;
(b1 \eq b2) => leftMultiplicity(addLink(assoc1, a2, b2), b1) = 1 + leftMultiplicity(assoc1, b1);
~(b1 \eq b2) => leftMultiplicity(addLink(assoc1, a2, b2), b1) = leftMultiplicity(assoc1, b1);

rightMultiplicity(void(i), a1) = 0;
(a1 \eq a2) => leftMultiplicity(addLink(assoc1, a2, b2), a1) = 1 + leftMultiplicity(assoc1, a1);
~(a1 \eq a2) => leftMultiplicity(addLink(assoc1, a2, b2), a1) = leftMultiplicity(assoc1, a1);

Taking association formal description, it can be realized that specific sorts for the set of instances of the connected classes need to be predefined. For the previous translations, this is the case for SetA and SetB. The generic sort for specifying the set of instances of a class is described in appendix A.

3.2.3 Composition Translation

✍ **Aggregation and Composition: Syntax and Semantics**
Associations can be in the form of aggregations meaning that objects of one class of the association are consisted of instances of the other class. This kind of association is known as "whole/part" relationship. Aggregations can be shared aggregations or compositions. Shared aggregations are merely conceptual and do not carry extra semantics comparing to plain associations. They are used just to show that conceptually the classes are not at the same hierarchical level.

Composition in turn is a strong form of aggregation that determines a dependency of the lifetime of the parts in respect to the whole. In a composition, the part cannot be shared by several wholes. In [UML99] it is stated that:

"Composite Aggregation is a strong form of aggregation which requires that a part instance be included in at most one composite at a time, although the owner may be changed over time." (Association – pg. 2-54)

The dependent lifetime is determined by [UML99]:

"Furthermore, a composite implies propagation semantics (i.e., some of the dynamic semantics of the whole is propagated to its parts)." (Association – pg. 2-54)

This propagation of dynamic semantics implies that the whole manages the creation and deletion of its parts. Moreover if the whole is copied or deleted, so need to be the parts as well. This propagation of semantics could be represented through message sending between classes. However these dynamic aspects are not yet treated in the semantic framework in its actual stage.

A composition in UML is represented as depicted in figure 5.

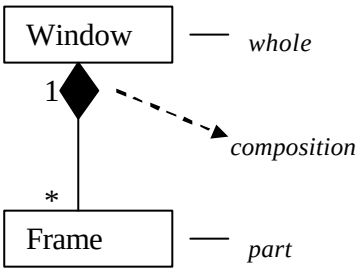

Figure 5. UML composition representation

✍ **Considerations on the Semantics**

The semantics for composition is not clear in UML as well as for aggregations. Aggregations as explained in UML documents are not more than plain associations with some conceptual value, being not a powerful characterization of a relationship.

Composition in turn establishes a strong form of relationship. Taking the fact that a part strongly belongs to its whole and that the whole manages its parts, a part could be seen as encapsulated in the whole instance in such a way that visibility to it could only be achieved through the whole avoiding side-effects in the system. However UML does not fairly state composition semantics.

Taking the static concerns, the main points that can be stated for composition semantics are:

- ✍ multiplicity at the whole side must be 1 at maximum;
- ✍ a part instance cannot be part in more than one composite at a time;
- ✍ instance reflexivity must be forbidden; i.e. a part cannot be part of itself;
- ✍ recursion must be stopped in any situation, a part cannot be part of a whole that is in turn its part.

In [UML99] it is assumed that:

"Both kinds of aggregations define a transitive, antisymmetric relationship (i.e., the instances form a directed, non-cyclic graph)." (Core – pg. 2-55)

However considering that aggregations are merely conceptual and do not determine dependency between the instances, only for compositions this assumption will be stated. Compositions do encompasses specific semantics and therefore some extra mapping rules need to be specified.

✍ **Mapping Rules**

Considering a composition Comp between generic classes A and B, being A the part and B the whole, the following mapping rules take place.

1. An operation to state that A is part of B is included in the association specification.
   isPartOf : Comp, A, B -> Bool

2. The semantics of this operation is expressed through the following axioms.
   isPartOf(addLink(comp1, a1, b1), a2, b2) => (a1 \eq a2) /\ (b1 \eq b2);
   ~(isPartOf(void(i), comp1, a1, b1));

3. An axiom to state that a part instance cannot belong by composition to more than one composite is written.
   ~(b1 \eq b2) => (isPartOf(comp1, a1, b1) /\ ~(isPartOf(comp1, a1, b2))) \/ (~(isPartOf(comp1, a1, b1) /\ (isPartOf(comp1, a1, b2)));

To complete composition semantics, some axioms in a more global level are added:

4. Considering compositions Comp1 from A to B and Comp2 from A to C in which B and C represent the wholes, the following axioms are written to match the semantics stating that an instance part cannot be part in more than one composite.

   assert
   ? **a : A, b : B, c : C, comp1: Comp1, comp2: Comp2**

   isPartOf(comp1, a, b) => ~(isPartOf(comp2, a, c));
   isPartOf(comp2, a, c) => ~(isPartOf(comp1, a, b));

5. Considering two compositions Comp1 and Comp2 between classes A and B, the following axioms are added to guarantee that recursion is stopped.
   assert
   ? **a : A, b : B, comp1: Comp1, comp2: Comp2**
   (isPartOf(comp1, a, b)) => ~(isPartOf(comp2, b, a));

6. Considering a composition Comp1 from A to A, the following axiom states that instance reflexivity is forbidden.
   assert
   ? **a : A, comp1: Comp1**
   ~(isPartOf(comp1, a, a));

3.2.4 Constraints Translation

In the algebraic semantic framework, some UML association constraints are translated, as: XOR, subset and derived.

✍ **XOR Constraint**
XOR is a constraint that can be established between two associations with the same source class. Taking the following example in figure 6, it can be realized that an exclusive or becomes necessary. An account can be of a Person or of a Company but not of both at the same time.

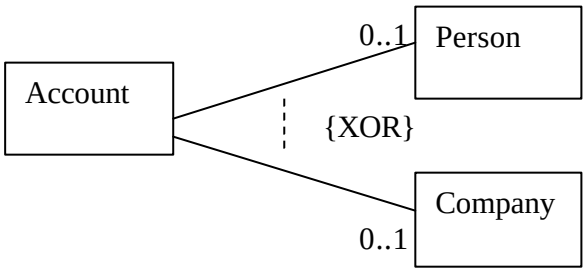

Figure 6. UML XOR constraint

Anytime an XOR constraint is encountered, an operator and axiom are generated to state the semantics determined by it. Considering a constraint XOR between generic associations from A to B and from A to C, the following mapping rules take place.

1. Operator defined to specify the constraint XOR between two associations.
   rightXOR : AssocAB, AssocAC -> Bool

2. Axiom stating the semantics of the constraint.
   assert
   rightXOR(assocAB, assocAC) =
   (~(isRightLinked(assocAB, a)) /\ (isRightLinked(assocAC, a))) \/
   ((isRightLinked(assocAB, a)) /\ ~(isRightLinked(assocAC, a)))

   Meaning that each A instance must be linked to a B or a C instance at a time.

✍ **Subset Constraint**

A subset constraint between two associations establishes a dependency from one in respect to the other. It can be clear in the example depicted in figure 7.

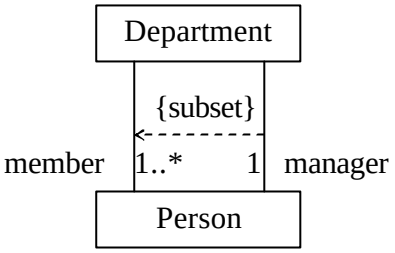

Figure 7. UML subset constraint

In this example, the constraint subset is used between the two associations to imply that a manager of a department must be one of its members.

To state that anytime there is a subset constraint, the multiplicity of the dependent association end must be less or equal the multiplicity of the association end in which it depends on, the following operator and axiom are written.

Consider two associations between generic classes A and B in which AssocAB2 depends on AssocAB1 at the right side.

1. Operator defined to specify the constraint subset between the two associations.
   rightSubset : AssocAB1, AssocAB2 -> Bool

2. Axiom stating the semantics of the constraint.
   assert
   rightSubset(assocAB1, assocAB2) =>
   ~(size(allRightLink(assocAB2, a)) > size(allRightLink(assocAB1, a)))

Following these steps, any constraint involving associations are possible to be formalized in the semantic framework.

3.2.5 Association Class Translation

✍ **Association Class Syntax and Semantics**

In an association between two classes, the association itself might have properties. In the UML, this is modelled as an association class, which is a modelling element that has both association and class properties. Because of its features inherited from classes and associations, it was feasible to treat also association classes in the framework in its first approach. Association classes are rendered in UML as depicted in figure 8.

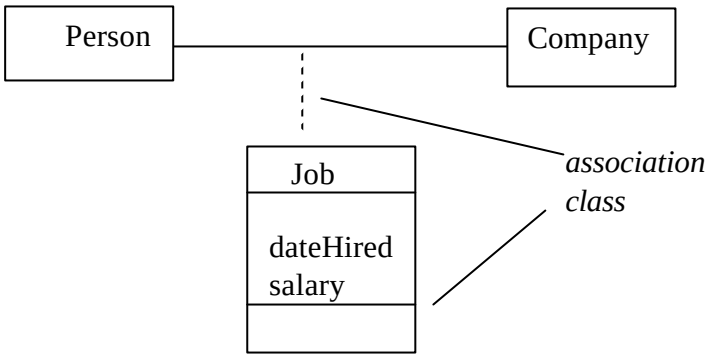

Figure 8. UML association class representation

✍ **Mapping Rules**
Considering an association class AsClassAB defined in association Assoc1 between classes A and B, the following mapping rules take place.

1. As in Classes, a single generator (or constructor) is considered:
   newAsClassAB : Nat, A, B, T1, …, Tn -> AsClassAB
   In which Nat is the type for association class identity (as used for associations in Association Translation – see rule 1), A and B the types representing the classes connected through the association and T1, ..., Tn the types of the association attributes.

2. Association class identity operations are written.
   identity : AsClassAB -> Nat
   __ \eq __ : AsClassAB, AsClassAB -> Bool
   In which association class equality is achieved through identity equality.

3. Two attributes are described.
   left : AsClassAB -> A
   right : AsClassAB -> B

4. Arguments of the constructor newAsClassAB expressing the attributes are described through the functions:
   getAssocAttr1 : AsClassAB -> T1
   setAssocAttr1 : AsClassAB , T1 -> AsClassAB

5. A mapping to the corresponding association is defined.
   assoc_AsClassAB : ->AssocAB

6. The set of existing instances is expressed through another sort, which here is called Sons. The operators defined in this Sons ADT are as follows:
   empty : -> Sons (generator or constructor of the empty set of instances)
   add : AsClassAB, Sons -> Sons (another constructor)
   first : Sons -> AsClassAB (first instance in the set)
   rest : Sons -> Sons (restant set)
   __\isIn__ : AsClassAB, Sons -> Bool

7. Then the relation between the instances of the association class and the links of the association is defined through an axiom.
   assert
   newAsClassAB(i, a1, b1) \isIn sons_AsClassAB <=>
   isLinked(assocAsClassAB, a1, b1)

   In which a1 and b1 are variables representing instances of A and B respectively.

3.2.6 Generalization Translation

✍ **Generalization Syntax and Semantics**

Generalizations are a kind of relationship in which one general thing is specialized in some specific ones. It is known as "is-a-kind-of" relationship in which a specialized thing is-a-kind-of a more general one. The general thing is called the superclass and the more specific things are called subclasses. The subclasses inherit all features of the super, including behavioral and structural features. Subclasses also inherit participation in associations from the superclass. The subclasses may even add new structure and behavior. The most important aspect concerning generalizations is that the instances of the subclass may be used anywhere an instance of the superclass is applied, but the reverse is not true. This is coherent concerning to the concept of subtyping.

Another important point in generalizations is that the subclass can even change the behavior of the parent. It can have an operation with the same signature as an operation in the parent but with a different implementation, what is called overridden. Through overridden polymorphism is achieved.

Generalization is represented in UML as depicted in figure 9.

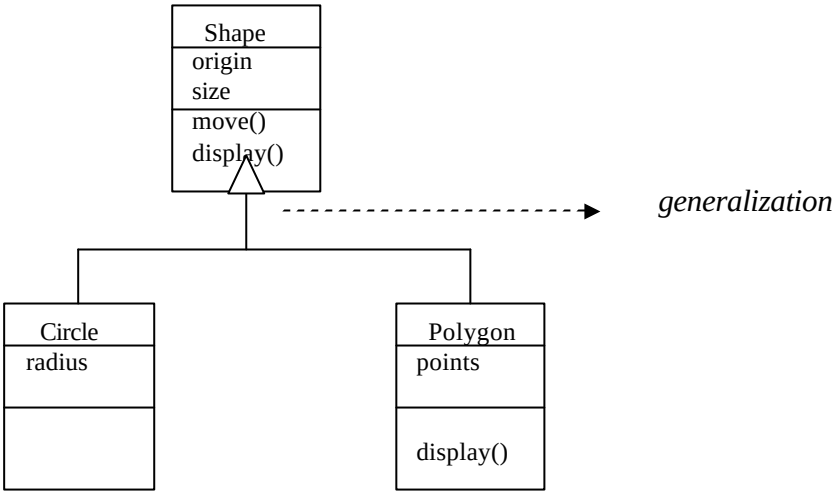

Figure 9. UML generalization representation

✍ **Mapping Rules**

Considering two generic classes A and B, in which B is subclass of A, the following mapping rules for inheritance can be established.

1. In the constructor of type B, the attributes of superclass A need also to be considered through the following function signature.
   newB : IdB, TA1, …, TAn, TB1, …, TBn, -> B
   In which IdB is the identity for type B, TA1, …, TAn represent the A instance attribute types, and TB1, …, TBn express the B instance attribute types.

2. To establish inheritance of structure a correspondence between the arguments of the generators of the ADTs is determined through the following axiom.
   assert
   PAB((newB(idb, attrA1, …, attrAn, attrB1, …, attrBn,), (newA(ida, attr1, …, attrn))) = equal(attrA1, attr1) /\ … /\ equal(attrAn, attrn)

   This axiom states that PAB, a structural projection from B to A, is determined through the correspondence between the common argument types of the generators of the super and sub classes. The first argument types of the B generator correspond to the inherited attributes.

3. To determine inheritance of associations, a simulate function as in the modular algebraic approach of section 2.5.2 is used.
   simulate : B -> A

4. Considering an association between the superclass A and a generic class C, the following axiom determines the semantics for association inheritance.

assert
? **c: C, a: A**
addLink(assocAC, a, c) = addLink(assocAC, (simulate(newB(id1, attr1,…, attrn)), c);

Inheritance of behavior is not yet provided in the framework.

3.3 Conclusion and Summary

The algebraic semantic framework in its actual stage encompasses the formal specifications for the main UML static model elements, as: Classes, Associations (including Compositions) and Generalizations. It also covers the formal specifications for some other static constructs of the UML Class Diagram as Association Classes and Association Constraints. Some other static building blocks of UML can also be incorporated in future by extension as Interfaces, variations of Classes (Abstract and Template), other forms of relationships (dependencies and realizations) as well as OCL constraints in the model that can also be translated.

Since all the static aspects are formalized, UML dynamic concerns can also be treated. As stated before, dynamic aspects are really powerful to express side-effect in operations through message sending between objects.

From the formal specifications generated, proofs can be applied over the models and therefore inconsistencies are checked. In future, basing in the formal specifications already achieved, transformations of models can be proved and rapid prototyping from design to code can be implemented.

In order to make clear the final resultant formal specifications for each UML static model element considered, the main mapping rules with their result are depicted in the following tables.

3.3.1 Summary Tables

The rules and axioms which numbers are pointed out in these tables can be found in the corresponding section of the translation from the UML model element to ADT.

Class Semantics	*Formal Operator*	*Rules and Axioms*
1. Single Generator	**newA : IdA, T1,…,Tn -> A**	**1**
2. Instance Attribute Descriptors	**getAttr1 : A -> attr1Type** **setAttr1 : A, attr1Type -> A**	**2, 6**
3. Instance Operations	Extensions of the operations in items 1 and 2	**3**
4. Object Identity	**identity : A -> IdA** **__\eq__ : A, A -> Bool**	**4,7,8**
5. Class Attributes	**getClassattr1 : -> T1** **setClassAttr1 : T1 -> T1**	**12**
6. Class Operations	Extensions of the operations in item 5.	**13**

Table 2 – Formal Specifications for Classes

Association Semantics	*Formal Operator (or axiom)*	*Rules and Axioms*
1. Generator of an Empty Set	**void : Nat -> Assoc1**	**1**
2. Generation of Links (simulating association instantiation)	**addLink : Assoc1, A, B -> Assoc1**	**2**
3. Stating that an Association cannot contain twice the same link.	**(a1 \eq a2) /\ (b1 \eq b2) =>** **addLink(addLink(assoc1, a1, b1), a2, b2) = addLink(assoc1, a1, b1);**	**10**
4. Deletion of Links	**removeLink : Assoc1, A, B -> Assoc1**	**3**
5. Return the set of Links for an instance of a classifier connected through the association.	**allLeftLink : Assoc1, B -> SetA** **allRightLink : Assoc1, A -> SetB**	**4, 12**

6. Multiplicity constraints established through the application of size operation of sets over allLeftLink and allRightLink.	**size(allRightLink(assoc1, a1)) = 1;** **~(size(allRightLink(assoc1, a1)) > 1);** **~(size(allRightLink(assoc1, a1)) < 1);**	**11**

Table 3 – Formal Specifications for Associations

Composition Semantics	*Formal Operator (or axiom)*	*Rules and Axioms*
1. State that an instance is part of another.	**isPartOf : Comp, A, B -> Bool**	**1**
2. A part instance cannot belong by composition to more than one composite.	**~(b1 \eq b2) => (isPartOf(comp1, a1, b1) /\ ~(isPartOf(comp1, a1, b2))) \/ (~(isPartOf(comp1, a1, b1) /\ (isPartOf(comp1, a1, b2)));** **isPartOf(comp1, a, b) => ~(isPartOf(comp2, a, c));** **isPartOf(comp2, a, c) => ~(isPartOf(comp1, a, b));**	**3, 4**
3. Recursion must be stopped and instance reflexivity forbidden.	**(isPartOf(comp1, a, b)) => ~(isPartOf(comp2, b, a));** **~(isPartOf(comp1, a, a));**	**5,6**

Table 4– Formal Specifications for Compositions

Generalization Semantics	*Formal Operator (or axiom)*	*Rules and Axioms*
1. In the constructor of the subclass, the attributes of the superclass need also to be considered.	**newB : IdB, TA1, …, TAn, TB1, …, TBn, -> B**	**1**

2. To establish inheritance of attributes, a correspondence between the arguments of the generators of the super and sub class ADTs is determined.	**assert** **PAB((newB(idb, attrA1, …, attrAn, attrB1, …, attrBn,), (newA(ida, attr1, …, attrn))) =** **equal(attrA1, attr1)** /\ **…** /\ **equal(attrAn, attrn)**	**2**
4. Inheritance of associations are expressed through a simulate function.	**assert** ? **c: C, a: A** **addLink(assocAC, a, c) =** **addLink(assocAC,** **(simulate(newB(id1, attr1,…, attrn)),** **c);**	**3, 4**

Table 5 – Formal Specifications for Generalizations

Chapter Four

Technologies Supporting the Semantic Framework

In this chapter the tools and technologies used to automate the generation of the formal specifications from a CASE tool are explained. In this context, the Graphtalk metatool is used to build the CASE tool, C++ is used to program the mapping rule functions and Larch Prover interprets the formal specifications generated to conduct validations on them. Each of these technologies and their integration are explained as follows.

4.1 The Practical Context to apply the Framework

In order to allow automatic generation of the formal specifications from a CASE tool based on the mapping rules described (see section 3.2), some technologies and tools are used in a suitable integrated way. First, the Graphtalk metatool was used to generate a CASE tool for the UML. From the user model built in the CASE tool, ASCII files containing the formal specifications following Larch Prover syntax are generated. This generation is automated through a Dynamic Linked Library (DLL) built in C++ from which functions can be called by Graphtalk CASE tool. The C++ source code invokes Graphtalk API (Application Programming Interface) functions in order to be able to access Graphtalk repositories of information from which all the information about the user model can be recovered.

Larch Prover ends this process by interpreting the formal specifications in the generated files being able to prove properties and detect inconsistencies about the models. Figure 10 shows a scheme of the integration among these different technologies.

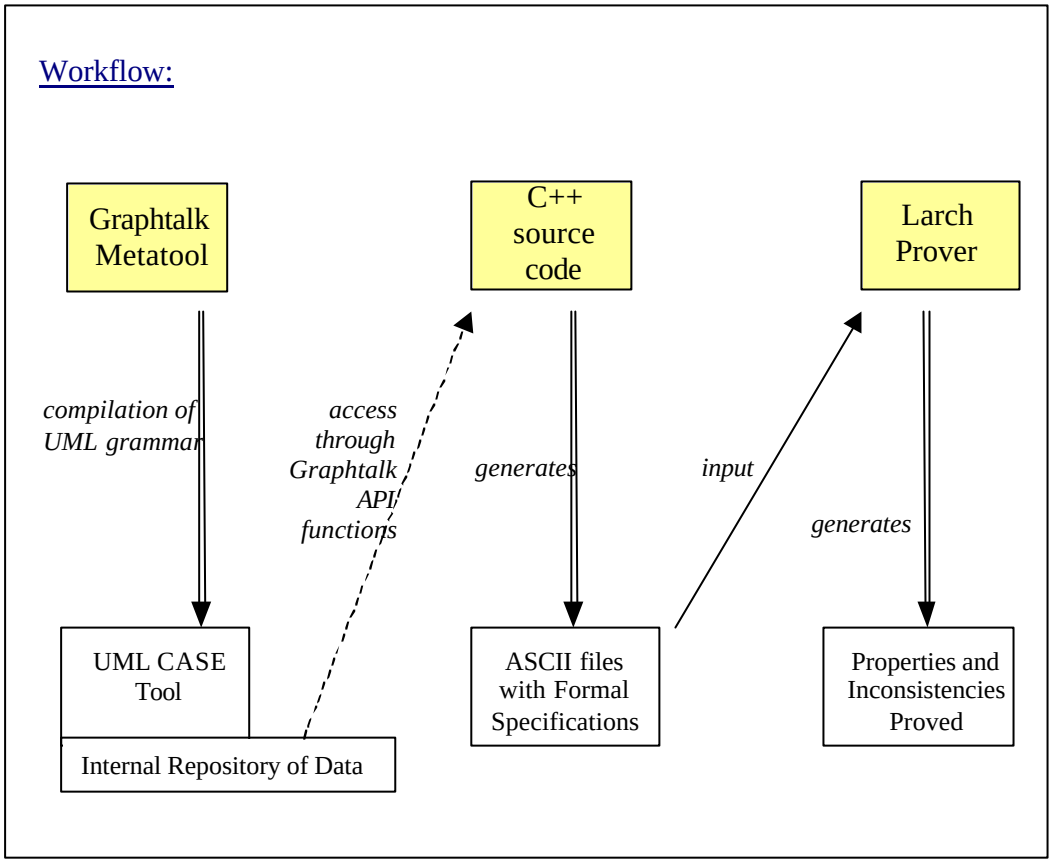

Figure 10 . Workflow integrating different technologies

In the next sections, each one of these technologies is described.

4.2 The Larch Prover

Larch itself is not in fact a language but an approach to define formal specifications being composed by a family of languages and tools. Larch Prover (LP) [GG89], the theorem prover of the Larch family is a set of proving tools that includes: rewriting, critical pair computation, Knuth-Bendix completion, proof by induction, proof by contradiction, and proof by case. LP has simple syntax and semantics, allows the definition of algebraic specifications to describe Abstract Data Types, and allows using rewrite rules to prove properties.

Larch Prover is based on Larch Shared Language (LSL). LSL is a two-tier language of the Larch family which has a top tier that is a behavioral interface specification language (BISL) tailored to a specific programming language, and a bottom tier that is used to describe the mathematical vocabulary used in the pre- and post-condition specifications. Besides the fact that LP is based on LSL it can also uses its own input syntactic format to the formal specifications that is the one followed in this work.

LP allows defining existential propositions (with the \E prefix), universal propositions (prefix \A) and propositions with usual logical connectors. It also supports first order predicate calculus with equality. The main principle behind LP is the rewrite process: each rule defined by an axiom is rewritten based on an operation in a process that goes until it can be concluded (terminated) or some inconsistency can be detected.

The complete command of LP uses a well-known algorithm: the Knuth-Bendix completion algorithm. This algorithm computes all the critical pairs and adds them in the system. The process stops with an inconsistency, which implies that the system is not consistent. Sometimes the process terminates without inconsistency. Otherwise the system does not terminate. The use of LP to proceed to proofs will be presented in chapter 5, section 5.4.

Other important aspects about LP are that it does not support genericity nor partial algebras and the only predefined type is Boolean. The semantics of the LP operations is expressed in axioms written through equations determining equality between terms.

4.3 The Graphtalk Metatool

To allow the automatic generation of formal specifications from a UML static model, a UML CASE tool was developed in Graphtalk carrying the mapping rules integrated in its context (see section 3.2). This integration is supported by a DLL built in C++ that provides the link edition from the C++ functions to Graphtalk. Therefore Graphtalk can invoke these functions.

Graphtalk allows both: work on the metalevel in order to generate CASE tools, and manipulate the tools generated at the model level. The work started at the metalevel specification.

4.3.1 Graphtalk Metamodel Level

Graphtalk metamodel is provided with an own meta-modeling language. To generate the CASE tool for the UML covering just the static diagram for the moment, it was necessary to describe the semantics of the UML static model elements in Graphtalk using its meta-modeling language. The following steps were taken in order to create the modeling tool:

- First: description of the specifications of the UML-tool in the meta-modeling language of Graphtalk was provided. Specifications in Graphtalk are stored in a file with .gti extension, e.g. UML.gti.

- Second: compilation of the source of the UML-tool (UML grammar) was performed obtaining a file with .gtm extension, e.g. UML. gtm. Using this .gtm file the developer can start creating his models.

The first step stated needs to be taken for each kind of diagram provided by the modeling (or CASE) tool. As the work presented here covers just the UML Static Diagram for the moment, only the semantics of this diagram was described to compose the UML grammar in Graphtalk.

Each diagram in Graphtalk is viewed as a graph. To describe a graph, it is necessary to work with four separate diagrams with complementary meanings:

- The ***semantics specification diagram*** defines all Graphtalk nodes and the links between these nodes. The nodes and links are used respectively to represent UML classes, and UML associations and generalizations.

- The ***property assignment diagram*** defines properties that are applied to the elements defined in the previous diagram. For example, the name of a class and the multiplicity of an association can be viewed as properties of the node representing a class and the link representing an association respectively.

- The ***shape specification diagram*** allows a graphical form to be created for the elements. A UML class, for example, gets its graphical representation in this diagram.

- The ***widget specification diagram*** allows widgets or other visual components to be defined to the CASE tool.

Figure 11 shows at the left side the first window of Graphtalk pointing out these diagrams from which the user starts working.

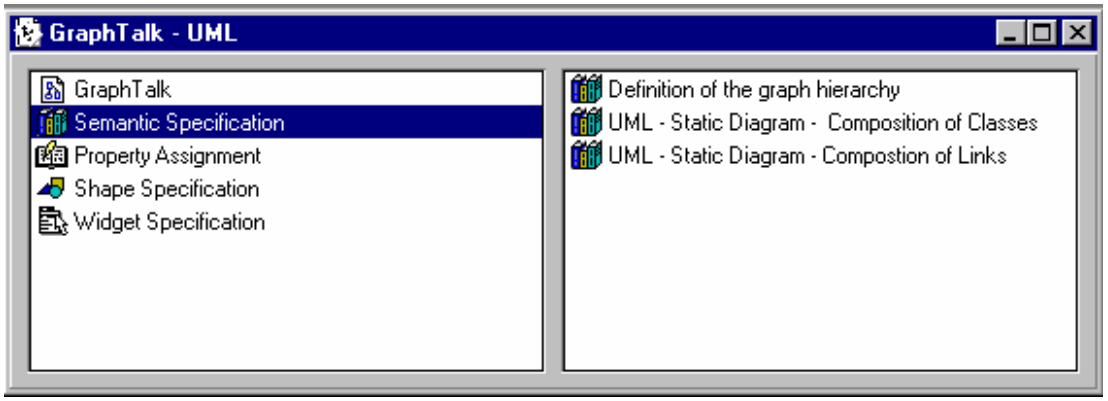

Figure 11. Four steps to define a graph.

These diagrams taken into account different Graphtalk elements. In the ***semantics specification diagram*** the following elements are used.

✍ **Nodes**

The concept of node in Graphtalk is similar to the concept of classes in an object-oriented language such as C++, Java or Smalltalk. A node represents an entity that can be instantiated. In the case of the UML static diagram, they are used to specify the classes as can be seen in figure 12. Nodes represent the elements that can be instantiated in an instance of the graph, i.e. in an instance of the static diagram at the model level.

✍ **Links**

Links are elements, which allow instances of nodes to be linked to each other. A graph can contain several different types of links. For the UML static diagram, links are used to represent associations (including aggregations and compositions) and generalizations as depicted in figure 12.

✍ **Entities**

An entity is an element, which has a meaning only inside Graphtalk. It is an abstract element that cannot be instantiated, i.e. the elements that are modeled in Graphtalk using an entity are not visible in the modeling tool. The role of an entity is to generalize other elements. The same entity can be an abstraction for graphs, nodes and links. A set of properties that is valid for a set of elements can be assigned to an entity which is the abstraction of these elements. An entity here is used to generalize the properties of associations, compositions and aggregations as shown in figure 12.

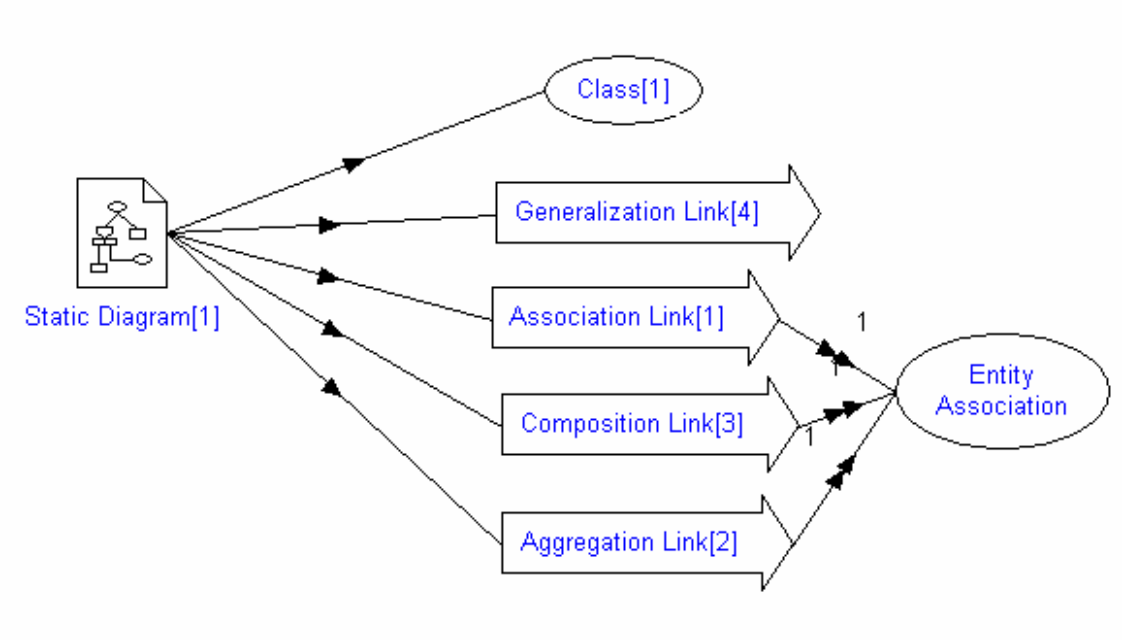

Figure 12. Semantic specification window for the UML static diagram

Considering that classes in UML have a list of attributes and a list of methods and that attributes and methods also have properties, a new local graph defining specific nodes needs to be defined as part of the static diagram graph (see figure 13). It is a local graph because these concepts are local to classes.

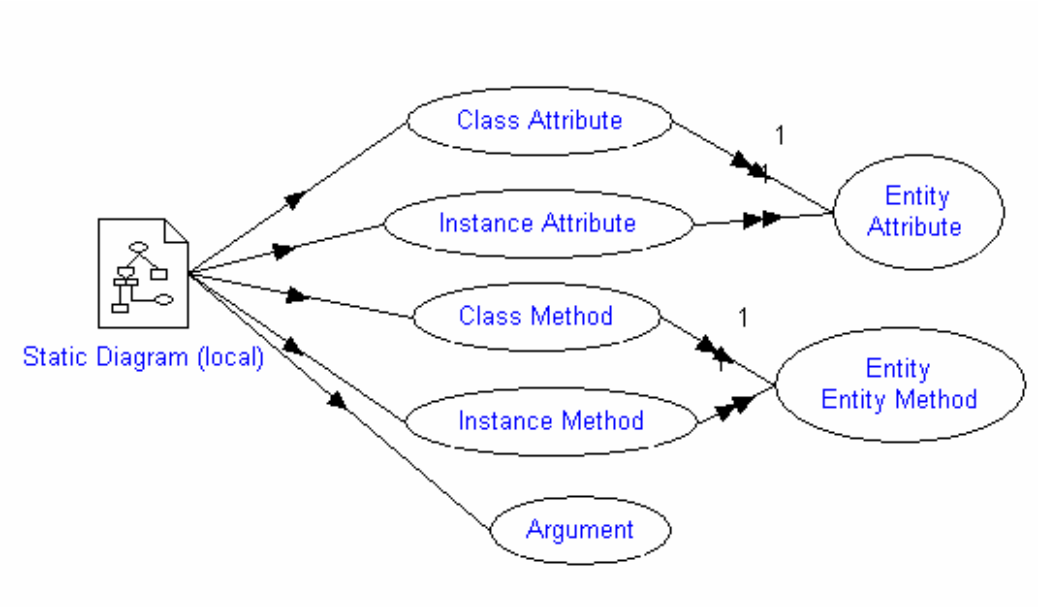

Figure 13. Semantics of a class

Argument node is defined to represent the arguments of methods as can be seen in figure 13.

Properties for these elements are defined in the ***property assignment diagram*** as can be seen in figure 14 for class properties. Properties are elements that will contain a value in the instance of the graph at the model level (i.e. in a user design). In Graphtalk properties can be of different types: Text, Boolean, List, Subnode, Popup Menu, etc.

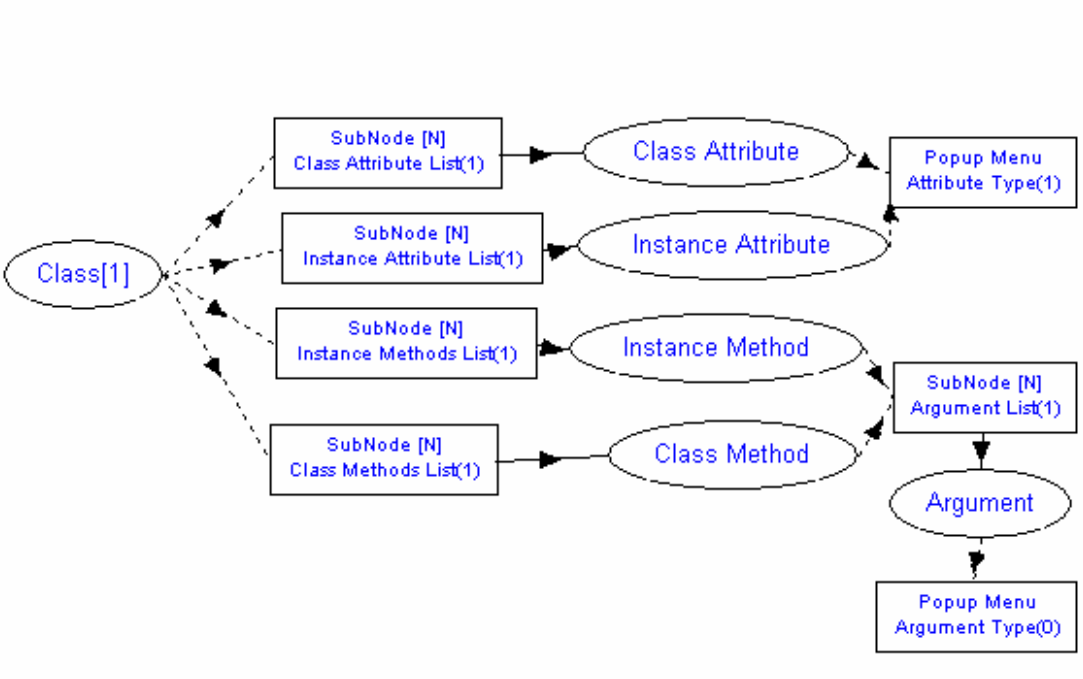

Figure 14. Properties of classes

Shapes to the model elements are given through the ***shape specification diagram***. The shapes given to each model element in the CASE tool can be viewed in the UML static diagram presented in chapter 5, section 5.1.

Finally, widgets were used in this work to allow the definition of a menu through which the developer can invoke the generation of the formal specifications from the CASE tool. To allow the link between Graphtalk and C++ the name of the C++ module (DLL) and the name of the invoked C++ function were provided in the widget specification.

Returning to figure 11, it can be clear the semantic diagrams defined for the UML static diagram (a graph in Graphtalk).

4.3.2 Graphtalk Model Level

At the model level, the developer is able to build his static diagrams in the CASE tool generated.

To start working, the developer needs to "run" the compiled source (UML.gtm). It is done just by creating a new file starting from the UML.gtm file. The user models will be saved in a file with extension .gti.

Using a modeling tool in Graphtalk means instantiating an hypergraph, i.e. making an instance of the modeling tool. The hypergraph in Graphtalk represents the entire modeling tool. When it is instantiated, the grammar describing graphs (diagrams) of the modeling language will also be instantiated.

Since the user completes his design, he can choose a menu option to ask for the generation of the algebraic Formal Specifications as can be seen in figure 15. The C++ function correspondent is then activated and processes the translations in order to get the files expressing the Abstract Data Types to be interpreted by Larch Prover.

Figure 15. The menu option to run the translation from UML to ADT

4.4 The use of C++ Programming Language

To program in C++ the environment chosen was the Microsoft Developer Studio for Microsoft Visual C++ 5.0.

The following steps were taken to build the Dynamic Linked Library of functions in C++:

- First, it was necessary to start a project of type *Win32 Dynamic-Link Library*. Visual C++ defines then a workspace with the same name of the project with reference to the project, e.g. **umltoadt.dsw** (*Project Workspace*) and **umltoadt.dsp** (*Project File*).
- Then, three files need to be defined:
  - **umltoadt.def** : the definition of the library with the name of the C++ function invoked by Graphtalk (**umltranslation**);
  - **umltoadt.h** : declares the signatures of all the functions to be used in the translation process by the C++ program; **umltranslation** is the main function that starts invoking the others;
  - **umltoadt.cpp** : the C++ source code in which the translating functions which signatures are declared in the **umltoadt.h** are programmed. The translating functions are programmed expressing the mapping rules for each UML model element described in section 3.2.

In the C++ source code, ASCII files are generated with the formal specifications resultant from the translation process. It is generated one ASCII file for each UML model element of the design done in the CASE tool. It is also generated one file for each additional type needed (as explained in section 3.2).

The fonts of the **umltoadt.def, umltoadt.h** and **umltoadt.cpp** files are presented in appendix B.

4.5 Conclusion

In this chapter it was reported how the translation process from UML to algebraic specifications describing ADTs could be automated. The work realized to this automation took into account the integration of Graphtalk and C++ in a suitable way. In the past there was already a project [JRG98] developed by students at Ecole des Mines de Nantes, France that made use of these technologies. The subject of the project was *Ré-ingénierie des systèmes classiques vers des systèmes à objets*, or in English *Reengineering of classical systems to build object oriented systems*. A transformation of designs done in Merise to OMT modeling language was defined. This project was used as the basis to the development of the C++ source code and to perform its integration to the Graphtalk.

Chapter Five

A Concrete Application of the Semantic Framework

In order to demonstrate how the translation process from UML to algebraic specifications describing ADTs works in practice, a UML static diagram developed in the CASE tool built in Graphtalk is presented. It is important to state that the environment in which the CASE tool runs is also the Graphtalk. The files containing the formal specifications generated for the static diagram are mentioned in this chapter and shown in appendix C.

These files will then be interpreted in Larch Prover. Therefore the results that can be obtained making use of the semantic framework are the properties and inconsistencies that Larch Prover can prove about the system. Some of them will also be described in this chapter.

This chapter ends with some conclusions that can be taken after putting the semantic framework to be used in practice.

5.1 The UML Static Diagram designed in the CASE tool

The CASE study chosen in order to demonstrate a UML Static Diagram drawn in the CASE tool generated from Graphtalk is a Library system. It considers the classes: Library, Publication, Copy, User, Teacher, Student, Loan and LocalUse. Loan characterizes the situation in which the user takes a copy to use out off the library, while LocalUse characterizes the internal use of copies by users. Considering it is an academic library, two main groups of users are defined: Teacher and Student. Each Publication may have any number of copies in the library. The corresponding UML Static Diagram is depicted in figure 16.

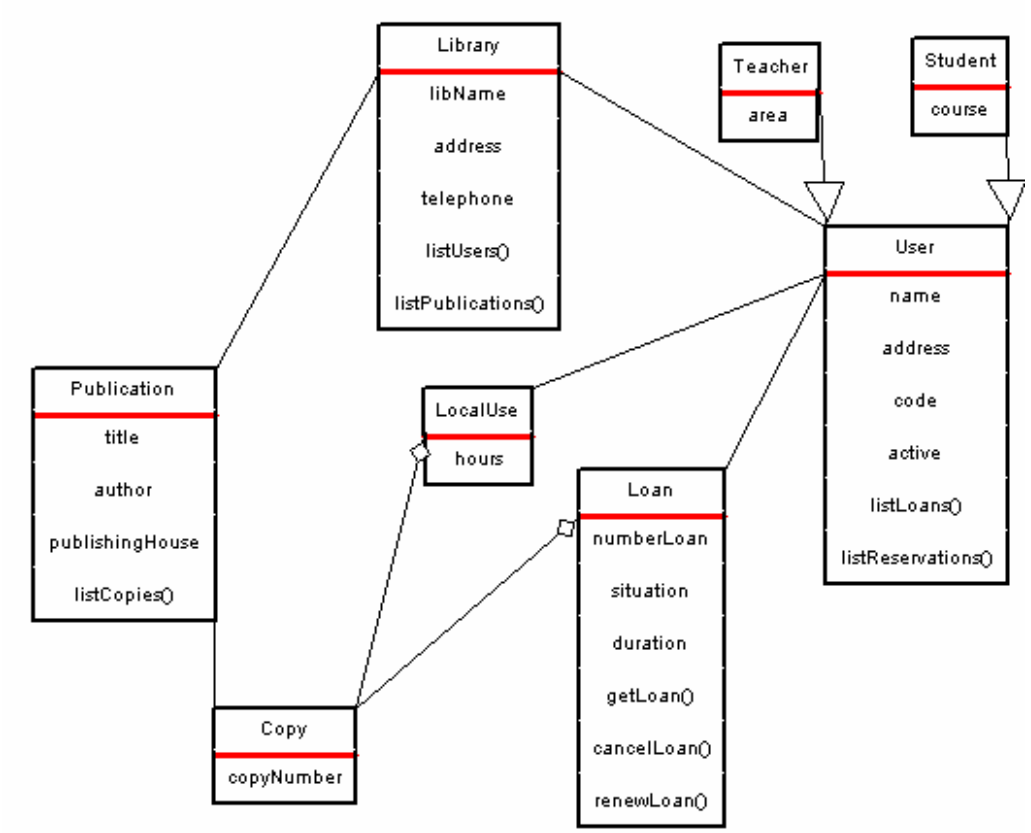

Figure 16. UML Static Diagram drawn in the Graphtalk CASE tool

As it was not so easy to use Graphtalk and the manuals were not so good, many improvements are still needed in the tool. The multiplicities of the associations, for example, are not explicitly shown in the diagram. They are described as follows:

- association from Publication to Library: Many (0 or plus) to Just One;
- association from Library to User: Just One to Many (0 or plus);
- association from Publication to Copy: Just One to One or more;
- composition from Copy to LocalUse (the target end is a composite aggregation): One or more to Just One;
- the same for the composition from Copy to Loan: One or more to Just One;
- association from LocalUse to User: Many (0 or plus) to Just One;
- in the same way association from Loan to User: Many (0 or plus) to Just One.

5.2 Formal Specifications generated for the UML Static Diagram

Asking for the generation of the formal specifications for this UML Static Diagram will result in the following ASCII files containing the algebraic specifications:

- one file per Class description: **Library.lp, Publication.lp, Copy.lp, LocalUse.lp, Loan.lp, User.lp, Teacher.lp** and **Student.lp**;

- one file describing a type Id for the objects of each type (or class) according to the rules in section 3.2.1: **IdLibrary.lp, IdPublication.lp, IdCopy.lp, IdLocalUse.lp, IdLoan.lp, IdUser.lp, IdTeacher.lp** and **IdStudent.lp**;

- one file expressing a type set for each class associated to another according to the rules explained in section 3.2.2: **SetLibrary.lp, SetPublication.lp, SetCopy.lp, SetLocalUse.lp, SetLoan.lp** and **SetUser.lp**;

- one file per each plain association and composition which name is composed by the three first letters of each class: **PubLib.lp** (association from Publication to Library), **LibUse.lp** (association from Library to User), **PubCop.lp** (association from Publication to Copy), **CompCopLoc.lp** (composition from Copy to LocalUse), **CompCopLoa.lp** (composition from Copy to Loan), **LoaUse.lp** (association from Loan to User), **LocUse.lp** (association from LocalUse to User);

- one file per generalization: **genUseTea.lp** (generalization between User and Teacher) and **genUseStu.lp** (generalization between User and Student).

These files, as explained before in chapter 4, will be interpreted by Larch Prover (LP) following then LP syntax. Therefore their extensions must be .lp. The description of some of these files is given in appendix C.

5.3 Checking Inconsistencies

Taking the previous UML Static Diagram describing a library system, an inconsistency can be detected by running the system formal described in LP. It is explained in the following section.

Another example of inconsistency still related to the use of composition is done in section 5.3.2.

5.3.1 A Composition Inconsistency

Taking the compositions between Copy and Loan and between Copy and LocalUse (see section 5.1), it can be realized that there is an inconsistency concerning multiplicities: multiplicity is Just One in both composites (Loan and LocalUse). According to the semantics of composition, a part instance cannot belong by composition to more than one whole at a time. The following composition axioms will determine a rule in LP that will not be respected:

assert
? **c: Copy, l : Loan, lu : LocalUse, c1: CompCopLoa, c2: CompCopLoc**

isPartOf(c1, c, l) => ~(isPartOf(c2, c, lu));
isPartOf(c2, c, lu) => ~(isPartOf(c1, c, l));

It will generate an error when the total system runs in LP since the multiplicity just one in Loan and LocalUse implies: ? **c: Copy, l : Loan, lu : LocalUse, ispartOf(c1, c, l)** ∧ **ispartOf(c2, c, lu)**.

5.3.2 A Composition with Generalization Inconsistency

Another example of inconsistency that can be detected in LP is depicted in figure 17:

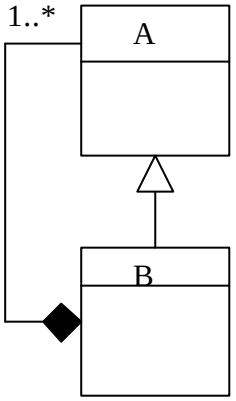

Figure 17. Instance reflexivity

Assuming that an instance cannot be part of itself (see section 3.2.3) what can cause circular specifications, the following axioms can be used in LP to try to write proofs over this example stating that it is inconsistent.

assert
? **a : A, b : B, comp: CompAB**

~(isPartOf(comp, a, a));
(isPartOf(comp, a, b)) => ~(isPartOf(comp, b, a));

5.4 Inconsistency with Constraint: a Concrete example of Proof written in LP

Taking the XOR constraint and its rules explained in section 3.2.4 (item XOR Constraint), the following example in figure 18 can be proven inconsistent through LP:

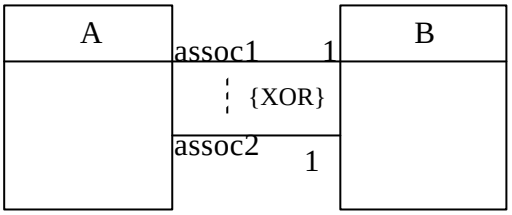

Figure 18. An inconsistency with XOR constraint

A complete command in LP for XOR constraint according to the rules (in bold) that were explained in section 3.2.4 is as follows.

% Assoc1 and Assoc2 types are simply expressed by the existential proposition P1
assert
\E Xas1 \E Yas1 (~(Xas1 \eq Yas1) /\ (rightMultiplicity(Xas1, a) = 1) /\ (rightMultiplicity(Yas1, a)= 1)) /\ rightXOR(Xas1, Yas1))

%the constraint XOR is defined by proposition P2
% -------- constraint rightXOR
declare operator **rightXOR : assoc1, assoc2 -> Bool**
assert ~(Xas \eq Yas) =>rightXOR(Xxas, Yas) = **(~(isRightLinked(Xas, a)) /\ (isRightLinked(Yas, a))) \/ ((isRightLinked(Xas, a)) /\ ~(isRightLinked(Yas, a)))**

%One simple example of proof is done
prove rightMultilicity(Xas, a) = 1 => isRightLinked(Xas, a)
  res by ind on Xas
     <> basis subgoal
     [] basis subgoal
     <> induction subgoal
  res by case a1 \eq a %proof by case
      <> case a1c \eq ac
      [] case a1c \eq ac
      <> case ~(a1c \eq ac)
      res by => %proof by implication
         <> => subgoal
                %addition of a trivial lemma
         assert identity(ac) \eq identity(a1c) = false
  crit as* with as* %critical pair computation
  [] => subgoal
  [] case ~(a1c\eq ac)
  [] induction subgoal
[] conjecture
qed %the proof is done

A critical pair is a potential ambiguity in a set of rules. It can be either a new fact forgotten in the system or an irremediable inconsistent fact. This kind of proof is generally not automatic, an expert user must choose the way to do it. Now it can be illustrated that the UML sample model in figure 18 is not consistent:

fix Yas1 as assoc2(a), Xas1 as assoc1(a) in P1 %elimination of \E

```
instantiate Xas by as1(a), Yas by as2(a) in P2          %elimination of \A
```

LP says that the system becomes inconsistent

5.5 Conclusion

In this chapter it was demonstrated practical examples of the translations and inconsistencies that can be checked. It was also demonstrated the use of LP to prove inconsistencies in the formal specifications.

Chapter Six

Conclusion

The formalization of Object Oriented analysis and design modeling languages has been claimed as a means to allow rigorous analysis, software comprehension and to guarantee consistency in all software development phases. The rigor imposed by formalization can also support early detection of errors in the development process what avoids that errors are carried till the implementation of the systems.

Even though UML is adopted as the standard Object Oriented modeling language for analysis and design it is not yet formalized.

The thesis of this research has been that formalizing UML through the use of a formal abstract language and also giving support to proceed to checks and validations on the formalized models can bring several contributions to software engineering and reengineering processes. Moreover formalization makes many ambiguities in the semantics arise being able to help in solving them.

6.1 Contributions

The main contribution of this work is to provide a basis to achieve a final UML formalization approach that can be used to support software engineering as well as software reengineering efforts. Formalization plays an important role in software engineering and reengineering environments in the sense that it can help in guaranteeing consistency in many stages: among model elements used in a model, between diagrams used to model a system, and between design and implementation through the refinement of models into code (and in the other way around: recovering design from code). Moreover it can contribute towards the specification of a final and unambiguous semantics to UML model elements.

In the semantic framework proposed in this thesis, the main concrete advantage taken is the early detection of errors that can be achieved in the analysis and design phases considering the software development life cycle. Avoiding that errors are carried till the source code is really cost effective since errors in the implemented system require really more effort and high cost to be eliminated.

In the context explained, many other contributions can be provided in future having the semantic framework as a basis:

- Improving OO legacy systems can be based on formal specifications in order to preserve semantics. Transformations of models based on refinements steps can be performed based on formal proved transformations.

- ✍ The formal specifications can make the link between design and implementation. Rapid prototyping generating source code from formal specifications has more chances to make it suitable to the system requirements.

- ✍ Ambiguities in UML semantics are solved through formalization.

- ✍ System quality and consistency are proved through the application of proofs in the formal specifications generated.

6.2 Future Work

In this thesis it was presented a first approach of a UML formalization method that has being developed making use of algebraic specifications to describe ADTs.

In the semantic framework presented in this dissertation, because of the limited time available to its development, only some static model elements of UML are formal described. Concerning the Static Diagram of UML, other model elements (or variations of them) are still to be considered in the formalization. It is considered as the main elements to continue with this work: Interfaces, Dependencies, Abstract Classes, Realizations and Constraints written in OCL.

Moreover it is considered the core semantics concerning each model element. Many other points can be considered in order to extend the framework:

- ✍ Extensions to the core concepts described are needed in order to have complete semantics specifications for the Structural Aspects of UML.

- ✍ Formalization of the remaining UML static model elements needs to be considered.

- ✍ Dynamic aspects of UML are also necessary to be formal described to have a complete description of elements semantics. Collaborations between objects are the first point to cover in order to complete some aspects of the semantics, such as to show the propagation of the dynamic semantics from the whole to its parts in a composite relationship (see chapter 3, section 3.2.3).

- ✍ Model transformations need to be formal proved. This is one of the most important points to achieve with formalization. Through proved transformations, reengineering and forward engineering efforts encompassing model refinements can be supported.

In fact, the main point to consider now is how the results of the proofs and checks obtained in LP can be demonstrated in the CASE tool to allow end user direct access.

As there was a real time constraint in order to develop this semantic framework, many of these points suggested as future extensions are still under investigation by the collaborators of this work. It is hoped that these extensions as soon as they are achieved,

they can be published and widely spread through the interested software engineering and academic community.

References:

[ADV99] Verónica Argañaraz, Ilse Dierickx, and Aline Vasconcelos. *A Pattern Representation Tool with UML.* EMOOSE – European Master of Science in Object Oriented Software Engineering. Ecole des Mines de Nantes, France. Vrije Universiteit Brussel (VUB), Belgium. February 1999.

[BRJ99a] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide.* Rational Software Corporation. Copyright ? 1999 by Addison Wesley Longman, Inc.

[BRJ99b] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language Reference Manual*. Rational Software Corporation. Copyright ? 1999 by Addison Wesley Longman, Inc.

[CE97] Tony Clark and Andy Evans. *Foundations of the Unified Modeling Language.* In NFM97: 2nd BCS-FACS Northern Formal Methods Workshop, Ilkley, UK, September 1997.

[CHS+97] Ercüment Canver, Friedrich von Henke, Detlef Schwier, Marie-Claude Gaudel, Nicolas Guelfi, Olivier Biberstein, Didier Buchs. *Comparison of Object-Oriented Formal Methods.* Universität Ulm 1997.

[CP99] Miro Casanova Paes . *Formal Representation of UML.* EMOOSE – European Master of Science in Object-Oriented and Software Engineering Technologies. Ecole des Mines de Nantes, France. Vrije Universiteit Brussel (VUB), Belgium. February 1999.

[CS97a] Computer Sciences Corporation. Graphtalk v3.5 User Manuel. 1997.

[CS97b] Computer Sciences Corporation. Graphtalk v3 API manual volume 1. 1997.

[CS97c] Computer Sciences Corporation. Graphtalk v3 API manual volume 2. 1997.

[EBFLR98] A.Evans, J-M. Bruel, R. France, K. Lano, and B. Rumpe. *Making UML Precise.* OOPSLA'98 Conference on object-Oriented Programming Systems, Languages, and Applications. Vancouver, October 1998.

[FELR97] Robert France, Andy Evans, Kevin Lano, and Bernhard Rumpe. *The UML as a Formal Modeling Notation.* OOPSLA'97 Workshop on Object-oriented Behavioral Semantics, p. 75-81. Atlanta, Georgia, USA, October 1997.

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

[GG91] Stephen J. Garland and John V. Guttag. *A Guide to LP, the Larch Prover.* MIT Laboratory for Computer Science, December 1991.

[HHK98] Ali Hamie, John Howse, Stuart Kent. *Modular Semantics for Object-Oriented Models*. Proceedings of Northern Formal Methods Workshop, eWics Series, Springer Verlag. September 1998.

[HJ95] I. Houston and M. Josephs. *The OMG's Core Object Model and compatible extensions to it.* Computer Standards and Interfaces, vol 17, nos 5 – 6, 1995.

[HR87] Horst Reichel. *Initial Computability Algebraic Specifications and Partial Algebras*. International Series of Monographs on Computer Science No. 2. Oxford Science Publications – 1987.

[JRG98] David Jaillet, Thierry Roussel, Nicolas Grelier. *Projet Transversal D'Informatique* – Ré-ingénierie des systèmes classiques vers des systèmes à objets. Ecole des Mines de Nantes, France. Fi95-info. April, 1998.

[LB98] K. Lano and J. Bicarregui. *Semantics and Transformations for UML Models.* UML'98 International Workshop. Mulhouse, France. June, 1998.

[MEY88] Bertrand Meyer. *Object-Oriented Software Construction*. International Series in Computer Science. Prentice Hall, 1988.

[MW93] M. Ward. *"Abstracting a Specification from Code"*. Journal of Software Maintenance: Research and Practice, vol 5, 1993, pp. 101- 122.

[PA99] Pascal André. *On Formalism in Object-Oriented Methods – the Object Identity Problem.* MSF – IRIN – Université de Nantes. April 1999. Internal Document.

[RAC94] Jean-Claude Royer, Pascal André, Dan Chiorean. *Object Design with Formal Classes.* MSF – IRIN – Université de Nantes. April 1994.

[Royer99a] Jean Claude Royer. *Abstract Data Types and Formal Classes.* IRIN – Université de Nantes. April 1999.

[Royer99b] Jean Claude Royer. *UML and ADT: A First Approach to Semantics and Verifications.* IRIN – Université de Nantes. June 1999. Internal Document.

[UML99] OMG Unified Modeling Language Specification. *UML Semantics*. Version 1.3. January 1999.

```
%%%%%%%%%%%%%%%%%%%%
% Entiers avec ordre <
% et minimum
% 5/1/99
% Nat1.lp
% systeme convergent

set name nat

declare sort Nat
declare variables i, j, k: Nat
declare operators
  0 :          -> Nat
  1 :          -> Nat
  2 :          -> Nat
  3 :          -> Nat
  4 :          -> Nat
  5 :          -> Nat
  s       : Nat      -> Nat
  __+__   : Nat, Nat -> Nat
  __-__   : Nat, Nat -> Nat
  inf   : Nat, Nat -> Bool
  equal   : Nat, Nat -> Bool
  ..

assert
  sort Nat generated by 0, s;

  1 = s(0);
  2 = s(1);
  3 = s(2);
  4 = s(3);
  5 = s(4);

  0 + i = i;
  i + 0 = i;
  s(i) + j = s(i + j);
  i + s(j) = s(i + j);

  0 - j = 0;
  s(i) - 0 = s(i);
  s(i) - s(j) = i - j;

  inf(0, 0);
  ~(inf(s(i), 0));
  inf(0, s(j));
  inf(s(i), s(j)) = inf(i, j);

  ~equal(0, s(i));
  ~equal(s(i), 0);
  % 0 \eq 0; OK mais ca utile
  equal(i:Nat, i:Nat);
  equal(s(i), s(j)) = equal(i:Nat, j:Nat);


  ..
```

```
%  ac +;  on moins ca termine
% converge oui


%%%%%%%%%%%
% definition des chaines
% 1/7/99
% String.lp
%------------------------
% charger Char.lp, Nat.lp avant
ex Char
ex Nat1

set name string

declare sorts String
declare variables str, str1, str2 : String
declare operators
  empty :          -> String
  add :         Char, String      -> String
  concat : String, String -> String
  length : String -> Nat
  is_empty : String -> Bool
  __<__ : String, String -> Bool
 equal : String, String -> Bool
  ..

assert
   sort  String generated by empty, add;

   is_empty(empty);
   ~is_empty(add(car1, str));

   concat(empty, str) = str;
   concat(add(car1, str1), str) = add(car1, concat(str1, str));

   length(empty) = 0;
   length(add(car1, str1)) = 1+length(str1);

   equal(empty, empty);
   ~equal(add(car1, str1), empty);
   ~equal(empty, add(car1, str1));
   ~equal(add(car1, str1), add(car2, str2)) = equal(str1, str2);

   ~(empty < empty);
   ~(add(car1, str1) < empty);
   (empty < add(car1, str1));
   add(car1, str1) < add(car2, str2) = (precede(car1, car2) \/
(equal(car1, car2) /\ (str1 < str2)));


    ..

% sh n empty < add(a, add(b, empty))
% sh n add(a, add(b, empty)) < add(a, add(b, empty))
% sh n add(b, add(b, empty)) < add(a, add(b, empty))
% sh n add(a, add(a, empty)) < add(a, add(b, empty))
```

```
% sh n add(a, add(a, empty)) < add(a, empty)
% sh n add(a, empty) < add(a, add(b, empty))
% sh n add(a, add(b, empty)) < add(a, add(b, add(c, empty)))
% sh n add(a, add(b, empty)) < add(b, add(b, add(c, empty)))
% sh n add(a, add(b, empty)) < add(b, empty)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%specification of the identifier type for A instances
%larch file: IdA.lp
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

set name IdA

declare sorts IdA
declare variables : idA1, idA2 : IdA
declare operators
newIdA :   -> IdA
nextIdA : IdA  -> IdA
__\eq__ : IdA, IdA  -> Bool
..

assert
sort IdA generated by newIdA, nextIdA;
newIdA \eq newIdA;
~(newIdA \eq nextIdA(idA1));
~(nextIdA(idA1) \eq newIdA);
nextIdA(idA1) \eq nextIdA(idA2) = (idA1 \eq idA2);
..

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%specification of the type Set for A Class
%larch file: SetA.lp
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


set name SetA

declare sorts A, SetA, Nat
declare variables aA1, aA2 : A, setA1, setA2 : SetA
declare operators
{} :  -> SetA
{__} : A -> SetA
insert : A, SetA -> SetA
__\U__ : SetA, SetA -> SetA
__\in__ : A, SetA -> Bool
__\I__ : SetA, SetA -> Bool
size : SetA -> Nat
..

assert
sort SetA generated by {}, insert;
{aA1} = insert(aA1, {});
~(aA1 \in {});
aA1 \in insert(aA2, setA1) <=> (aA1 \eq aA2 \/ aA1 \in setA1);
{} \I setA1;
insert(aA1, setA1) \I setA2 <=> (aA1 \in setA2 /\ setA1 \I setA2);
```

```
aA1 \in (setA1 \U setA2) <=> (aA1 \in setA1 \/ aA1 \in setA2);
% axioms for size operator
size({}) = 0;
(aA1 \in setA1) => size(insert(aA1, setA1)) = size(setA1);
~(aA1 \in setA1) => size(insert(aA1, setA1)) = 1+size(setA1);
..
```

UMLTOAD.DEF

```
LIBRARY umltoadt
DESCRIPTION 'Demons for GraphTalk'

EXETYPE WINDOWS

DATA SINGLE   MOVEABLE
CODE MOVEABLE DISCARDABLE

HEAPSIZE 1024

EXPORTS
umltranslation
```

UMLTOAD.H

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>
#include <windows.h>
#include "gti.h"

using namespace std;

// ************************************************
// Functions used for the Mapping from UML to ADT
// ************************************************

extern "C" {
void umltranslation(OBJ viewer, OBJ hyperUML, OBJ arg);
// Main function for the generation of ADT from a UML model
// This function will be invoked by an action_item in a Graphtalk menu
// It operates on an instance of a UML Static Diagram
}


//*************************************************
//Functions to Generate the Lines in an Output Text File
//*************************************************
void frecordln (ostream& gen, string& str);
//Record strings in a text output file

void frecord (ostream& gen, string& str);
//Record strings in a text output file
//do not skip a line each time it records a new string

void fskipline (ostream& gen);
//skip a line in the output file


//*****************************************************************************
//To finish the declares operators and assert sections it necessary to write two dots.
//This is Larch Prover syntax.
//*****************************************************************************
void endSection(ostream& ffile);


//*************************************************
//Translation from UML Class to Abstract Data Type
//Translating Rule and Auxiliary Functions
//*************************************************

void TranslateClass(OBJ viewer, OBJ unGraphUML);
//This function will invoke all the mapping rule functions for Classes

//recover the number of attributes of a class
```

```
SHORT genAttrNumber(OBJ unGraphUML, OBJ unClassUML);

void rule1Class(ostream& fclass, string classname, OBJ unClassUML, SHORT nbAttr);
//Rule 1 generates the name of the sort being specified

void rule2Class(ostream& fclass, string classname, OBJ unClassUML, SHORT nbAttr);
//Rule 2 constructs the section declare variables, with
//two variables per instance attribute, two variables of the type
//being defined and one variable of type identity

void rule3Class(ostream& fclass, string classname, OBJ unClassUML, SHORT nbAttr);
//Rule 3 starts the declare operators section of the algebraic specification
//with the standard operations new, identity and object equality

void rule4Class(ostream& fclass, string classname, OBJ unClassUML, SHORT nbAttr);
//Rule4 generates the accessor operations for the attributes

void rule5Class(ostream& fclass, string classname, OBJ unClassUML);
//Rule 5 generates the constant operations and formaloperators
//the constants represent examples of instances of the class

void rule6Class(ostream& fclass, string classname, SHORT nbAttr, OBJ unClassUML);
//Rule 6 starts the axioms part
//The axioms are used to state the semantics of the operations
////and the constants.

void rule7Class(ostream& fclass, string classname, SHORT nbAttr, OBJ unClassUML);
//Rule 7 defines the axioms for the accessors

void rule9Identity(ostream& fidentity, string classname);
//constructs the type identity in another output file



//*********************************************************
//Translation from UML Associations to sorts in Larch Prover
//*********************************************************

void TranslateLinks(OBJ unClassUML);
//recover the links from a node class
//if the link is an association or aggregation, invokes tranlateassoc function;
//otherwise, invokes translatecomposite

void TranslateAssoc(OBJ unLinkUML, OBJ sourceClass, OBJ targetClass);
//Translation from a UML Association to a sort in Larch Prover

void rule1Assoc(OBJ unLinkUML, ostream& flink, string linkname, string sourcename, string
targetname);
//Rule 1 generates the name of the sort being specified
//and a comment defining the sort to be described

void rule2Assoc(ostream& flink, string linkname, string sourcename, string targetname);
//Rule 2 constructs the section declare variables, with
//two variables per object type, two variables per object type set,
//two variables of type equal the type being defined, and
//one variable of type Nat

void rule3Assoc(ostream& flink, string linkname);
//Rule 3 starts the declare operators section of the algebraic specification
//with the standard operations void, identity and association equality

void rule4Assoc(ostream& flink, string linkname, string sourcename, string targetname);
//Rule 4 declares the operators for add and remove links in the association

void rule5Assoc(OBJ unLinkUML, ostream& flink, string linkname, string sourcename, string
targetname);
//Rule 5 declares operators to return the set of links for a given instance,
//and also declares the operators to test if the association isEmpy and if
//two instances are linked

void rule6Assoc(ostream& flink, string linkname, string sourcename, string targetname);
```

```
//Rule 6 starts the axioms part
//The axioms are used to state the semantics of the operations.

void rule7Assoc(ostream& flink, string linkname, string sourcename, string targetname,
OBJ unLinkUML);
//Rule 7 states the axioms for multiplicity constraints

void rule8Assoc(ostream& flink, string linkname, string sourcename, string targetname);
//Rule 8 states the axioms for association identity

void rule9Set(ostream& fset, string classname);
//Rule 9 generates the sort Set for the source and target Class
//of the Association

//Special Rules for Compositions
string rule1Composition(string linkname);
string rule2Composition(string linkname, string sourcename, string targetname);
void rule3Composition(ostream& flink, string linkname, string sourcename, string
targetname);
void rule4Composition(ostream& flink, string linkname, string sourcename, string
targetname);
```

```
#include "umltoadt.h"
```

UMLTOAD.CPP

```
// ************************************************
// Functions to be used in the translation from UML -> ADT
// ************************************************


// Main function for the generation of ADT from a UML model
// This function will be invoked by an action_item in a Graphtalk menu
// It operates on an instance of a UML Static Diagram
void umltranslation(OBJ viewer, OBJ hyperUML, OBJ arg)
{
        //recover all the graphs associated to a hyper UML
        //in this version the only existent graph is the static, so only one graph
        //will be recovered for the list
        OBJ graphList, unGraphUML;

        graphList = GtiGraphsOf(hyperUML);

        while(graphList != NIL) {
                unGraphUML = GtiCar(graphList);
                if (GtiClassSymbol(GtiClassOf(unGraphUML)) == GtiSymbol("Static Diagram"))
{
                        TranslateClass(viewer, unGraphUML);

                }
                graphList = GtiCdr(graphList);
}
}

//*****************************************************
//Functions to Generate the Lines in an Output Text File
//*****************************************************

//Record strings in a text output file
//skip a line each time it records a new string
void frecordln (ostream& gen, string& str)
{
        gen << str << endl;
}

//Record strings in a text output file
//do not skip a line each time it records a new string
void frecord (ostream& gen, string& str)
```

```
{
	gen << str;
}

//skip a line in the output file
void fskipline (ostream& gen)
{
	gen << endl;
}


//*****************************************************************************
//To finish the declares operators and assert sections it necessary to write two dots.
//This is Larch Prover syntax.
//*****************************************************************************
void endSection(ostream& ffile)
{
	string fin;

	fin = "..";
	frecordln(ffile, fin);
}


//*************************************************
//Translation from a UML Class to a sort in Larch Prover
//*************************************************
void TranslateClass(OBJ viewer, OBJ unGraphUML)
{
	OBJ nodeListUML, unClassUML;
	SHORT numberAttr;
	//OBJ linkListUML, unLinkUML;

	string  filename, classname, fileidentity;

	nodeListUML = GtiNodesOf(unGraphUML);
	while(nodeListUML != NIL) {
		unClassUML = GtiCar(nodeListUML);
		if (GtiClassSymbol(GtiClassOf(unClassUML)) == GtiSymbol("Class")) {
			//generate the name of the text file for the class description
			classname = GtiNameOf(unClassUML);
			filename = classname + ".lp";

			//creates the file in write and text mode
			ofstream fclass(filename.c_str());

			//recover the number of instance attributes of a class;
			//this number is necessary to recover the attributes that will be
used
			//in some mapping rules
			numberAttr = genAttrNumber(unGraphUML, unClassUML);

			//calls the mapping rule functions to generate the
			//lines in the algebraic specification
			rule1Class(fclass, classname, unClassUML, numberAttr);
			rule2Class(fclass, classname, unClassUML, numberAttr);
			rule3Class(fclass, classname, unClassUML, numberAttr);
			rule4Class(fclass, classname, unClassUML, numberAttr);
			rule5Class(fclass, classname, unClassUML);
			//to finalize the declare operators section
			endSection(fclass);
			//starts the axiom section
			rule6Class(fclass, classname, numberAttr, unClassUML);
			//to finalize the assert section
			endSection(fclass);

			//constructs the type identity in another output file
			fileidentity = "Id" + classname + ".lp";
			//creates the file for sort Identity in write and text mode
			ofstream fidentity(fileidentity.c_str());
			rule9Identity(fidentity, classname);
```

```
                        //this type of file is closed automatically and doesn't require a
close command

                        //***********
                        //after finishing the translation for the Class, then
                        //it is invoked the translation for the links (associations,
compositions, aggregations)
                        //for this class
                        TranslateLinks(unClassUML);
                }
                nodeListUML = GtiCdr(nodeListUML); //withdraw the first element of the
list
        }
}

//Translating Rules from UML Class to Abstract Data Type

//Recover the number of attributes of a Class
//Attributes are a SubNode property which cerries a list of attributes
SHORT genAttrNumber(OBJ unGraphUML, OBJ unClassUML)
{       // Test if the first Node of the list is in fact a Class
        // In this version we only have classes as nodes, but
        // thinking about future enhancements it is better to keep this test

        //Obj unAttrUML;
        SHORT nbAttrs;

        //unAttrUML = GtiCar(GtiNodesOf(GtiCoreOf(unClassUML)));
        //if (GtiCar(GtiNodesOf(GtiCoreOf(unClassUML))) == unClassUML)
        //if (GtiClassSymbol(GtiClassOf(unAttrUML)) == GtiSymbol("Instance Attribute")) {
        nbAttrs = GtiSubSubNodesCount(unClassUML, GtiSymbol("Instance Attribute List"));
        return nbAttrs;

}

//Rule 1 generates the name of the sort being specified
//and a comment defining the sort to be described
void rule1Class(ostream& fclass, string classname, OBJ unClassUML, SHORT nbAttr)

{       string specname, comment, attrname, attrtypename;
        SHORT i, j;
        OBJ unAttributUML;
        string listattr[10];
        bool found;


        comment =
"%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%";
        frecordln(fclass, comment);
        comment = "%specification of the properties and behavior for the instances  of
class " + classname;
        frecordln(fclass, comment);
        comment = "%larch file: " + classname + ".lp";
        frecordln(fclass, comment);
        comment =
"%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%";
        frecordln(fclass, comment);
        fskipline(fclass);
        fskipline(fclass);
        //set name section starts the formal specification for the sort
        specname = "set name " + classname;
        frecordln(fclass, specname);
        fskipline(fclass);
        specname = "declare sorts " + classname;
        frecord(fclass, specname);
        //declares the identifier type
        specname = ", Id";
        frecord(fclass, specname);
        frecord(fclass, classname);
```

```
        //declares the type Set
        specname = ", Set" + classname;
        frecord(fclass, specname);

        //recover the types of the attributes to the section declare sorts
        for(i=0; i<nbAttr; i++) {
                unAttributUML = GtiGetSubSubNode(unClassUML, GtiSymbol("Instance Attribute
List"), i);
            attrname = GtiNameOf(unAttributUML);
                attrtypename = GtiNameOf(GtiGetMenu(unAttributUML, GtiSymbol("Attribute
Type")));
                found = false; //controls if the type already exists in the list
                //traverses the attribute types array - listAttr - and select the
attribute types
                //to compose the declare sorts section; each attribute type must be
declared
                //only once
                for (j=i; j>0; j--) {
                        if (attrtypename == listattr[j])
                                found = true;
                }
                if (!found)
                        specname = ", " + attrtypename;

                listattr[i] = attrtypename;
        }

        frecordln(fclass, specname);
        //delete listattr; //no garbage collection in C++
}

//Rule 2 constructs the section declare variables, with
//two variables per instance attribute, two variables of the type
//being defined and one variable of type identity
void rule2Class(ostream& fclass, string classname, OBJ unClassUML, SHORT nbAttr)

{       string specvariables, var1, var2, variablename, typeattr, sectionname;
        SHORT i, count;//the goal of the variable count is to allow just two variable
declarations per line
        OBJ unAttributUML;

        count = 0;
        specvariables = "declare variables ";
        var1 = "a" + classname + "1,";
        var2 = " a" + classname + "2";
        specvariables = specvariables + var1 + var2 + " : " + classname;
        frecord(fclass, specvariables);
        count = count + 1;
        for(i=0; i<nbAttr; i++) {
                unAttributUML = GtiGetSubSubNode(unClassUML, GtiSymbol("Instance Attribute
List"), i);
            variablename = GtiNameOf(unAttributUML);
                //generates two variables with the attribute type
            specvariables = variablename + "1,  " + variablename + "2 : ";
                //recover the type of the attribute
                typeattr = GtiNameOf(GtiGetMenu(unAttributUML, GtiSymbol("Attribute
Type")));
                if (count == 2)
                {   fskipline(fclass);
                        sectionname = "declare variables ";
                        frecord(fclass, sectionname);
                }
                else
                {       variablename = ", ";
                        frecord(fclass, variablename);
                }

                frecord(fclass, specvariables);
                frecord(fclass, typeattr);
                count = count + 1;
```

```
        }
        if (count == 2)
                {   fskipline(fclass);
                        specvariables = "declare variables ";
                        frecord(fclass, specvariables);
                };
        specvariables = ", id : Id";
        frecord(fclass, specvariables);
        specvariables = classname;
        frecordln(fclass, specvariables);
}

//Rule 3 starts the declare operators section of the algebraic specification
//with the standard operations new, identity and object equality
void rule3Class(ostream& fclass, string classname, OBJ unClassUML, SHORT nbAttr)

{       OBJ unAttributUML;
        string specoperators, typeName, comment;
        SHORT i;

        specoperators = "declare operators";
        fskipline(fclass);
        frecordln(fclass, specoperators);
        comment = "% generator";
        frecordln(fclass, comment);
        specoperators = "new" + classname + " : Id" + classname;
        for(i=0; i<nbAttr; i++) {
                specoperators = specoperators + ",";
                unAttributUML = GtiGetSubSubNode(unClassUML, GtiSymbol("Instance Attribute
List"), i);
                //recover the metaproperty type ofthe attribute
                //in Graphtalk the default value for this property needs to be
                //string
                typeName = GtiNameOf(GtiGetMenu(unAttributUML, GtiSymbol("Attribute
Type")));
                specoperators = specoperators + " " + typeName;
        }
        specoperators = specoperators + " -> " + classname;
        frecordln(fclass, specoperators); //generates the operation new
        fskipline(fclass);
        comment = "% operations for identity and object equality";
        frecordln(fclass, comment);
        specoperators = "identity :   " + classname + " -> " + "Id" + classname;
        frecordln(fclass, specoperators); //generates the operation that returns the
identity of an object
        specoperators = "__\\eq__ :  " + classname + ", " + classname + " -> " + "Bool";
        frecordln(fclass, specoperators); //generates the operation for
identity/functional equality
}

//Rule4 generates the accessor operations for the attributes
void rule4Class(ostream& fclass, string classname, OBJ unClassUML, SHORT nbAttr)
{
        OBJ unAttributUML;
        string accessor, typeName, attrName, comment;
        SHORT i;

        fskipline(fclass);
        comment = "% accessors for the instance variables";
        frecordln(fclass, comment);
        for(i=0; i<nbAttr; i++) {
                //recover the attribute in the instance variables list;
                //its name and type
                unAttributUML = GtiGetSubSubNode(unClassUML, GtiSymbol("Instance Attribute
List"), i);
                attrName = GtiNameOf(unAttributUML);
                typeName = GtiNameOf(GtiGetMenu(unAttributUML, GtiSymbol("Attribute
Type")));
                //the first accessor is the get, to recover a value of the attribute
related to
                //one object
```

```
                accessor = "get" + attrName + " : ";
                accessor = accessor + " " + classname + " -> ";
                frecord(fclass, accessor);
                frecordln(fclass, typeName);
                //the second accessor is the set, to change the value of the attribute to
one object
                accessor = "set" + attrName + " : ";
                accessor = accessor + " " + classname + ", " + typeName + " -> ";
                accessor = accessor + classname;
                frecordln(fclass, accessor);
        }
}

//Rule 5 generates the constant operations and formaloperators
//the constants represent examples of instances of the class
void rule5Class(ostream& fclass, string classname, OBJ unClassUML)
{
        string comment, constant, formalOpr, oprName;
        SHORT nbOprs, i;
        OBJ unOprUML;

        comment = "% constants to represent examples of instances";
        fskipline(fclass);
        frecordln(fclass, comment);
        constant = "one" + classname + " :   ";
        constant = constant + " -> " + classname;
        frecordln(fclass, constant);
        constant = "another" + classname + " :       ";
        constant = constant + " -> " + classname;
        frecordln(fclass, constant);
}


//Rule 6 starts the axioms part
//The axioms are used to state the semantics of the operations
//and the constants
void rule6Class(ostream& fclass, string classname, SHORT nbAttr, OBJ unClassUML)
{
        string comment, axiom, constantaxiom1, constantaxiom2, attrvariable, var1, var2;
        SHORT i;
        OBJ unAttributUML;

        comment = "% axioms";
        fskipline(fclass);
        frecordln(fclass, comment);
        axiom = "assert";
        frecordln(fclass, axiom);
        axiom = "sort " + classname + " generated by " + "new" + classname + ";";
        frecordln(fclass, axiom);
        comment = "%axioms for identity";
        frecordln(fclass, comment);
        axiom = "identity";
        frecord(fclass, axiom);
        axiom = "(new" + classname + "(id";
        constantaxiom1 = "one" + classname + "= new"+ classname + "((newId" + classname +
")";
        constantaxiom2 = "another" + classname + "= new"+ classname + "((nextId"+
classname + "(newId" + classname + "))";
        for(i=0; i<nbAttr; i++) {
                axiom = axiom + ",";
                constantaxiom1 = constantaxiom1 + ",";
                constantaxiom2 = constantaxiom2 + ",";
                unAttributUML = GtiGetSubSubNode(unClassUML, GtiSymbol("Instance Attribute
List"), i);
                attrvariable = GtiNameOf(unAttributUML);
                attrvariable = attrvariable + "1";
                axiom = axiom + attrvariable;
                constantaxiom1 = constantaxiom1 + attrvariable;
                constantaxiom2 = constantaxiom2 + attrvariable;
        }
        axiom = axiom + ")) = id;";
```

```
        constantaxiom1 = constantaxiom1 + ");";
        constantaxiom2 = constantaxiom2 + ");";
        frecordln(fclass, axiom);
        fskipline(fclass);
        comment = "%axioms for the constants";
        frecordln(fclass, comment);
        frecordln(fclass, constantaxiom1);
        frecordln(fclass, constantaxiom2);

        //axiom to state the semantics for object equality
        comment = "% axiom to state the semantics for object equality";
        frecordln(fclass, comment);
        var1 = "a" + classname + "1";
        var2 = "a" + classname + "2";
        axiom = var1 + " \\eq " + var2 + " = " + "identity(" + var1 + ")" + " \\eq" + "
identity(" + var2 + ");";
        frecordln(fclass, axiom);

        //call rule 7 that defines the axioms for the accesors
        rule7Class(fclass, classname, nbAttr, unClassUML);

}

//Rule 7 defines the axioms for the accessors
void rule7Class(ostream& fclass, string classname, SHORT nbAttr, OBJ unClassUML)
{
        string comment, axiom, axiom2, varstructure, varstructure2, mainattr, mainattr2,
attrname1, attrname2, attrvariable;
        SHORT i,j;
        OBJ unAttributUML;

        fskipline(fclass);
        comment = "% axioms to state the semantics of the attribute accessors";
        frecordln(fclass, comment);

        //for each instance attribute, two axioms are generated being each one to one
accessor

                for (i=0; i<nbAttr; i++) {
                        unAttributUML = GtiGetSubSubNode(unClassUML, GtiSymbol("Instance
Attribute List"), i);
                        attrname1     = GtiNameOf(unAttributUML);
                        mainattr     = attrname1 + "1";
                        mainattr2       = attrname1 + "2";
                        axiom = "get"+ attrname1 + "(" + "new" + classname + "(id";
                        axiom2 = "set"+ attrname1 + "(" + "new" + classname + "(id";
                        j = 0;

                        varstructure = "";
                        varstructure2 = "";
                        while (j < nbAttr)
                        {
                                unAttributUML = GtiGetSubSubNode(unClassUML,
GtiSymbol("Instance Attribute List"), j);
                                attrname2    = GtiNameOf(unAttributUML);
                                attrvariable = attrname2 + "1";
                                varstructure = varstructure + ", "+ attrvariable;
                                if (attrname1 == attrname2)
                                        attrvariable = attrname2 + "2";
                                else
                                        attrvariable = attrname2 + "1";
                                varstructure2 = varstructure2 + ", " + attrvariable;
                                j++;
                        }
                        axiom = axiom + varstructure + "))" + " = " + mainattr + ";";
                        frecordln(fclass, axiom);
                        axiom2 = axiom2 + varstructure + ")," + mainattr2 + ")";
                        axiom2 = axiom2 + " = " + "new" + classname + "(id" + varstructure2
+ ");";
                        frecordln(fclass, axiom2);
                }
```

```
}

//Rule 8 defines the axioms for the constants

//Rule 9 defines a type Identity specific to a class
void rule9Identity(ostream& fidentity, string classname) {

        string sortname, sentence, comment;

        sortname = "Id" + classname;

        comment = "%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%";
        frecordln(fidentity, comment);
        comment = "%specification of the identifier type for " + classname + " isntances";
        frecordln(fidentity, comment);
        comment = "%larch file: " + sortname + ".lp";
        frecordln(fidentity, comment);
        comment = "%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%";
        frecordln(fidentity, comment);

        fskipline(fidentity);
        sentence = "set name " + sortname;
        frecordln(fidentity, sentence);
        fskipline(fidentity);

        sentence = "declare sorts " + sortname;
        frecordln(fidentity, sentence);
        sentence = "declare variables : id" + classname + "1, ";
        sentence = sentence +  "id" + classname + "2" + " : " + sortname;
        frecordln(fidentity, sentence);
        sentence = "declare operators";
        frecordln(fidentity, sentence);
        sentence = "new" + sortname + " : " + "  -> " + sortname;
        frecordln(fidentity, sentence);
        sentence = "next" + sortname + " : " + sortname + "   -> " + sortname;
        frecordln(fidentity, sentence);
        sentence = "__\\eq__ : " + sortname + ", " + sortname + "  -> " + "Bool";
        frecordln(fidentity, sentence);
        endSection(fidentity);

        //axioms
        fskipline(fidentity);
        sentence = "assert";
        frecordln(fidentity, sentence);
        sentence = "sort " + sortname + " generated by new" + sortname + ", next" +
sortname + ";";
        frecordln(fidentity, sentence);
        sentence = "new"+ sortname + " \\eq " + "new" + sortname + ";";
        frecordln(fidentity, sentence);
        sentence = "~(new" + sortname + " \\eq " + "next" + sortname + "(id" + classname +
"1));";
        frecordln(fidentity, sentence);
        sentence = "~(next" + sortname + "(id" + classname + "1)" + " \\eq " + "new" +
sortname + ");";
        frecordln(fidentity, sentence);
        sentence = "next" + sortname + "(id" + classname + "1)" + " \\eq " + "next" +
sortname + "(id" + classname + "2)" + " = ";
        sentence = sentence + "(id" + classname + "1" + " \\eq id" + classname + "2);";
        frecordln(fidentity, sentence);
        endSection(fidentity);
}


//**********************************************************
//Translation from UML Associations to sorts in Larch Prover
//**********************************************************

//recover the links from a node class
//if the link is an association or aggregation, invokes tranlateassoc function;
```

```
//otherwise, invokes translatecomposite
void TranslateLinks(OBJ unClassUML)
{
        OBJ listLinkUML, unLinkUML, sourceClass, targetClass;
        string filename;

        listLinkUML = GtiFromLinksOf(unClassUML);

        //Recover the first link of the list
   //Recover the target node of the Link


        if (listLinkUML != NIL){
                unLinkUML = GtiCar(listLinkUML);
                sourceClass = GtiLinkOrg(unLinkUML);
                targetClass = unClassUML;

                while(listLinkUML != NIL) {
                        if (GtiClassSymbol(GtiClassOf(unLinkUML)) == GtiSymbol("Association
Link"))
                                TranslateAssoc(unLinkUML, sourceClass, targetClass);
                        if (GtiClassSymbol(GtiClassOf(unLinkUML)) == GtiSymbol("Composition
Link"))
                                TranslateAssoc(unLinkUML, sourceClass, targetClass);


                        listLinkUML = GtiCdr(listLinkUML);
                        unLinkUML = GtiCar(listLinkUML);
                }
                GtiDropList(listLinkUML);
        }
}


//Translation from a UML Association to a sort in Larch Prover
void TranslateAssoc(OBJ unLinkUML, OBJ sourceClass, OBJ targetClass)
{

        string  sourcename, targetname, linkname, filename;


        linkname = GtiGetString(unLinkUML, GtiSymbol("Name"));
        if (GtiClassSymbol(GtiClassOf(unLinkUML)) == GtiSymbol("Composition Link"))
                linkname = rule1Composition(linkname);
        filename = linkname + ".lp";

        //creates the file in write and text mode
        ofstream flink(filename.c_str());
        sourcename = GtiNameOf(sourceClass);
        targetname = GtiNameOf(targetClass);

        //the translation rules
    rule1Assoc(unLinkUML, flink, linkname, sourcename, targetname);
        rule2Assoc(flink, linkname, sourcename, targetname);
        rule3Assoc(flink, linkname);
        rule4Assoc(flink, linkname, sourcename, targetname);
        rule5Assoc(unLinkUML, flink, linkname, sourcename, targetname);
        //to finalize the declare operators section
        endSection(flink);

        //starts the axiom section
        rule6Assoc(flink, linkname, sourcename, targetname);
        rule7Assoc(flink, linkname, sourcename, targetname, unLinkUML);
        rule8Assoc(flink, linkname, sourcename, targetname);
        //to finalize the assert section
        endSection(flink);

        //constructs the type Set for each Class connected through the association;
        //each type Set will be generated in one LP file
        //first, the file Set for the source Class
    filename = "Set" + sourcename + ".lp";
```

```
        ofstream fset(filename.c_str());
        rule9Set(fset, sourcename);
        //after, the file Set for the target Class
    filename = "Set" + targetname + ".lp";
        ofstream fset1(filename.c_str());
        rule9Set(fset1, targetname);
}


//Rule 1 generates the name of the sort being specified
//and a comment defining the sort to be described
void rule1Assoc(OBJ unLinkUML, ostream& flink, string linkname, string sourcename, string
targetname)
{
        string specname, comment;

        comment =
"%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%";
        frecordln(flink, comment);
        if (GtiClassSymbol(GtiClassOf(unLinkUML)) == GtiSymbol("Composition Link"))
                comment = rule2Composition(linkname, sourcename, targetname);
        else
                comment = "%specification of the association " + linkname + " between
Classes: " + sourcename + " and " + targetname;
        frecordln(flink, comment);
        comment = "%larch file: " + linkname + ".lp";
        frecordln(flink, comment);
        comment =
"%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%";
        frecordln(flink, comment);
        fskipline(flink);
        fskipline(flink);

        //set name section starts the formal specification for the sort
        specname = "set name " + linkname;
        frecordln(flink, specname);
        fskipline(flink);

        //declares the association sort
        //declares the source class and target class as object types
        specname = "declare sorts " + linkname + ", " + sourcename + ", " + targetname;
        frecord(flink, specname);


        //declares the type Nat that is the type for the association identifier
        specname = ", Nat";
        frecord(flink, specname);

        //declares the types Set related to the Set of instances of the object types
associated
        specname = ", Set" + sourcename + ", Set" + targetname;
        frecordln(flink, specname);

}

//Rule 2 constructs the section declare variables, with
//two variables per object type, two variables per object type set,
//two variables of type equal the type being defined, and
//one variable of type Nat
void rule2Assoc(ostream& flink, string linkname, string sourcename, string targetname)

{       string specvariables, var1, var2;



        specvariables = "declare variables ";
        frecord(flink, specvariables);

        //variables for the type being defined
        var1 = linkname + "1, ";
        var2 = linkname + "2";
```

```
        specvariables = var1 + var2 + " : " + linkname + ", ";
        frecord(flink, specvariables);

        //variables for the source object type of the link
        var1 = "a" + sourcename + "1,";
        var2 = " a" + sourcename + "2";
        specvariables = var1 + var2 + " : " + sourcename + ", ";
        frecord(flink, specvariables);

        //variables for the target object type of the link
        var1 = "a" + targetname + "1,";
        var2 = " a" + targetname + "2";
        specvariables = var1 + var2 + " : " + targetname;
        frecordln(flink, specvariables);

        //skip line
        specvariables = "declare variables ";
        frecord(flink, specvariables);

        //variables for the Set of source objects type
        var1 = "Set" + sourcename + "1";
        var2 = ", Set" + sourcename + "2";
        specvariables = var1 + var2 + " : " + "Set" + sourcename + ", ";
        frecord(flink, specvariables);

        //variables for the Set of target objects type
        var1 = "Set";
        var1 = var1 + targetname + "1, ";
        var2 = "Set" + targetname + "2";
        specvariables = var1 + var2 + " : " + "Set" + targetname + ", ";
        frecord(flink, specvariables);

        //variable of type Natural for the association identity
        var1 = "i : Nat";
        frecordln(flink, var1);
}

//Rule 3 starts the declare operators section of the algebraic specification
//with the standard operations void, identity and association equality
void rule3Assoc(ostream& flink, string linkname)

{       string specoperators, comment;


        specoperators = "declare operators";
        fskipline(flink);
        frecordln(flink, specoperators);

        //operation void for the generation of an empty association
        comment = "%generator of an empty association";
        frecordln(flink, comment);
        specoperators = "void :       Nat     -> " + linkname;
        frecordln(flink, specoperators);

        //operations for association identity needed for Larch Prover
        comment = "%association identity";
        fskipline(flink);
        frecordln(flink, comment);
        specoperators = "identity :   " + linkname + "  -> Nat";
        frecordln(flink, specoperators);

        specoperators = "__\\eq__  :  " + linkname + ", " + linkname + "  -> Bool";
        frecordln(flink, specoperators);
}

//Rule 4 declares the operators for add and remove links in the association
void rule4Assoc(ostream& flink, string linkname, string sourcename, string targetname)

{       string specoperators, comment;

        comment = "%operators to create and remove links ";
```

```
        fskipline(flink);
        frecordln(flink, comment);

        //addlink
        specoperators = "addLink : ";
        specoperators = specoperators + linkname + ", " + sourcename + ", " + targetname;
        specoperators = specoperators + " -> " + linkname;
        frecordln(flink, specoperators);

        //removelink
        specoperators = "removeLink : ";
        specoperators = specoperators + linkname + ", " + sourcename + ", " + targetname;
        specoperators = specoperators + " -> " + linkname;
        frecordln(flink, specoperators);

}

//Rule 5 declares operators to return the set of links for a given instance,
//and also declares the operators to test if the association isEmpy and if
//two instances are linked; declares also the constants for the multiplicities
void rule5Assoc(OBJ unLinkUML, ostream& flink, string linkname, string sourcename, string
targetname)

{       string specoperators, comment;

        //tests if two instances are linked through the association
        comment = "%operator to test if two instances are linked";
        fskipline(flink);
        frecordln(flink, comment);
        specoperators = "isLinked : ";
        specoperators = specoperators + linkname + ", " + sourcename + ", " + targetname;
        specoperators = specoperators + " -> Bool";
        frecordln(flink, specoperators);

        //tests if one instance is linked through the association
        comment = "%operator to test if one instance is linked";
        fskipline(flink);
        frecordln(flink, comment);
        specoperators = "isLeftLinked : ";
        specoperators = specoperators + linkname + ", " + targetname;
        specoperators = specoperators + " -> Bool";
        frecordln(flink, specoperators);

        specoperators = "isRightLinked : ";
        specoperators = specoperators + linkname + ", " + sourcename;
        specoperators = specoperators + " -> Bool";
        frecordln(flink, specoperators);


        //tests if the association isEmpty
        comment = "%operator to test if the association is Empty";
        fskipline(flink);
        frecordln(flink, comment);
        specoperators = "isEmpty : ";
        specoperators = specoperators + linkname;
        specoperators = specoperators + " -> Bool";
        frecordln(flink, specoperators);

        //return the Set of instances of an object type linked to an instance
        //of the other object type connected through the association
        comment = "%operator to return the Set of instances linked to a given instance";
        fskipline(flink);
        frecordln(flink, comment);
        specoperators = "allLeftLink : ";
        specoperators = specoperators + linkname + ", " + targetname;
        specoperators = specoperators + " -> Set" + sourcename;
        frecordln(flink, specoperators);

        specoperators = "allRightLink : ";
        specoperators = specoperators + linkname + ", " + sourcename;
        specoperators = specoperators + " -> Set" + targetname;
```

```
        frecordln(flink, specoperators);

        //constants to the association multiplicities
        fskipline(flink);
        comment = "%operators for multiplicity values";
        frecordln(flink, comment);
        specoperators = "leftMultiplicity : " + linkname + ", " + targetname + " -> Nat";
        frecordln(flink, specoperators);
        specoperators = "rightMultiplicity : " + linkname + ", " + sourcename + "  ->
Nat";
        frecordln(flink, specoperators);

        if (GtiClassSymbol(GtiClassOf(unLinkUML)) == GtiSymbol("Composition Link"))
                rule3Composition(flink, linkname, sourcename, targetname);

}

//Rule 6 starts the axioms part
//The axioms are used to state the semantics of the operations.
void rule6Assoc(ostream& flink, string linkname, string sourcename, string targetname)
{
        string comment, axiom;

        comment = "% axioms";
        fskipline(flink);
        frecordln(flink, comment);
        axiom = "assert";
        frecordln(flink, axiom);
        axiom = "sort " + linkname + " generated by " + "void, addLink;";
        frecordln(flink, axiom);
        fskipline(flink);

        //axioms to state that an association cannot contain twice the same link
        comment = "%axiom to state that tuples of instance values cannot be equal in an
association";
        frecordln(flink, comment);
        axiom = "(a" + sourcename + "1" + " \\eq " + "a" + sourcename + "2)";
        axiom = axiom + " /\\ ";
        axiom = axiom + "(a" + targetname + "1" + " \\eq " + "a" + targetname + "2)" + "
=> ";
        frecord(flink, axiom);
        axiom = "addLink(addLink(" + linkname + "1" + ", a" + sourcename + "1, a" +
targetname + "1), ";
        axiom = axiom + "a" + sourcename + "2, " + "a" + targetname + "2)" + " = addLink("
+ linkname + "1, ";
        axiom = axiom + "a" + sourcename + "1, " + "a" + targetname + "1);";
        frecordln(flink, axiom);

        //axioms for the operation isEmpty
        fskipline(flink);
        comment = "%axioms for the isEmpty operation";
        frecordln(flink, comment);
        axiom = "isEmpty(void(i));";
        frecordln(flink, axiom);
        axiom = "~(isEmpty(addLink(" + linkname + "1, " + "a" + sourcename + "1, " + "a" +
targetname + "1)));" ;
        frecordln(flink, axiom);

        //axioms to test if two instances are linked - operator isLinked
        fskipline(flink);
        comment = "%state when two instances of object types are linked";
        frecordln(flink, comment);
        axiom = "~(isLinked(void(i),";
        axiom = axiom + "a" + sourcename + "1, " + "a" + targetname + "1));";
        frecordln(flink, axiom);
        axiom = "isLinked(addLink(" + linkname + "1, " + "a" + sourcename + "1, " + "a" +
targetname + "1),";
        axiom = axiom + "a" + sourcename + "2, " + "a" + targetname + "2) = ";
        axiom = axiom + "((a" + sourcename + "1 \\eq " + "a" + sourcename + "2";
        axiom = axiom + " /\\ " + "a" + targetname + "1" + " \\eq " + "a" + targetname +
"2)";
```

```
        axiom = axiom + " \\/ " + "isLinked(" + linkname + "1, " + "a" + sourcename + "2,
" + "a" + targetname + "2" + "));";
        frecordln(flink, axiom);

        //axioms to state the semantics of the operations allLefLink
        //and allRightLink in terms of addLink generator
        fskipline(flink);
        comment = "%state the semantics for the operations allLeftLink and allRightLink
through addLink generator";
        frecordln(flink, comment);
        axiom = "(a" + targetname + "1 \\eq a" + targetname + "2) => ";
        axiom = axiom + "allLeftLink(addLink(" + linkname + "1, a" + sourcename + "1, a" +
targetname + "2), a" + targetname + "1)";
        axiom = axiom + " = insert(a" + sourcename + "1, allLeftLink(" + linkname + "1, a"
+ targetname + "1));";
        frecordln(flink, axiom);

        axiom = "(a" + sourcename + "1 \\eq a" + sourcename + "2) => ";
        axiom = axiom + "allRightLink(addLink(" + linkname + "1, a" + sourcename + "2, a"
+ targetname + "1), a" + sourcename + "1)";
        axiom = axiom + " = insert(a" + targetname + "1, allRightLink(" + linkname + "1,
a" + sourcename + "1));";
        frecordln(flink, axiom);

        axiom = "~(a" + targetname + "1 \\eq a" + targetname + "2) => ";
        axiom = axiom + "allLeftLink(addLink(" + linkname + "1, a" + sourcename + "1, a" +
targetname + "2), a" + targetname + "1)";
        axiom = axiom + " = allLeftLink(" + linkname + "1, a" + targetname + "1);";
        frecordln(flink, axiom);
        axiom = "~(a" + sourcename + "1 \\eq a" + sourcename + "2) => ";
        axiom = axiom + "allRightLink(addLink(" + linkname + "1, a" + sourcename + "2, a"
+ targetname + "1), a" + sourcename + "1)";
        axiom = axiom + " = allRightLink(" + linkname + "1, a" + sourcename + "1);";
        frecordln(flink, axiom);

        //semantics for allLeftLink and allRightLink in terms of void generator
        fskipline(flink);
        comment = "%state the semantics for allLeftLink and allRightLink through void
generator";
        frecordln(flink, comment);
        axiom = "allLeftLink(void(i), a" + targetname + "1) = {}: Set" + targetname + ";";
        frecordln(flink, axiom);
        axiom = "allRightLink(void(i), a" + sourcename + "1) = {}: Set" + sourcename +
";";
        frecordln(flink, axiom);


        //axioms to state when an instance is linked
        fskipline(flink);
        comment = "%state when one instance is linked through the association";
        frecordln(flink, comment);
        axiom = "~(isLeftLinked(void(i), a" + targetname + "1));";
        frecordln(flink, axiom);
        axiom = "~(isRightLinked(void(i), a" + sourcename + "1));";
        frecordln(flink, axiom);


        axiom = "isLeftLinked(addLink(" + linkname + "1, a" + sourcename + "1, a" +
targetname + "1), a" + targetname + "2) = ((";
        axiom = axiom + "a" + targetname + "1" + " \\eq " + "a" + targetname + "2) \\/
isLeftLinked(";
        axiom = axiom + linkname + "1, a" + targetname + "2));";
        frecordln(flink, axiom);

        axiom = "isRightLinked(addLink(" + linkname + "1, a" + sourcename + "1, a" +
targetname + "1), a" + sourcename + "2) = ((";
        axiom = axiom + "a" + sourcename + "1" + " \\eq " + "a" + sourcename + "2) \\/
isRightLinked(";
        axiom = axiom + linkname + "1, a" + sourcename + "2));";
        frecordln(flink, axiom);
```

```
        //left and rightMultiplicity axioms
        comment = "%axioms for left and rightMultiplicity operators";
        fskipline(flink);
        frecordln(flink, comment);
        axiom = "leftMultiplicity(void(i), a" + targetname + "1) = 0;";
        frecordln(flink, axiom);
        axiom = "rightMultiplicity(void(i), a" + sourcename + "1) = 0;";
        frecordln(flink, axiom);

        axiom = "(a" + targetname + "1" + " \\eq " + "a" + targetname + "2) => ";
        axiom = axiom + "leftMultiplicity(addLink(" + linkname + "1, a" + sourcename + "2,
a" + targetname + "2),";
        axiom = axiom + "a" + targetname + "1) = leftMultiplicity( "+ linkname + "1, a" +
targetname + "1) + 1;";
        frecordln(flink, axiom);

        axiom = "~(a" + targetname + "1" + " \\eq " + "a" + targetname + "2) => ";
        axiom = axiom + "leftMultiplicity(addLink(" + linkname + "1, a" + sourcename + "2,
a" + targetname + "2),";
        axiom = axiom + "a" + targetname + "1) = leftMultiplicity( "+ linkname + "1, a" +
targetname + "1);";
        frecordln(flink, axiom);


        fskipline(flink);
        axiom = "(a" + sourcename + "1" + " \\eq " + "a" + sourcename + "2) => ";
        axiom = axiom + "rightMultiplicity(addLink(" + linkname + "1, a" + sourcename +
"2, a" + targetname + "2),";
        axiom = axiom + "a" + sourcename + "1) = rightMultiplicity( "+ linkname + "1, a" +
sourcename + "1) + 1;";
        frecordln(flink, axiom);

        axiom = "~(a" + sourcename + "1" + " \\eq " + "a" + sourcename + "2) => ";
        axiom = axiom + "rightMultiplicity(addLink(" + linkname + "1, a" + sourcename +
"2, a" + targetname + "2),";
        axiom = axiom + "a" + sourcename + "1) = rightMultiplicity( "+ linkname + "1, a" +
sourcename + "1);";
        frecordln(flink, axiom);


}

//Rule 7 states the axioms to recover the multiplicity values
//and axioms for multiplicity constraints
void rule7Assoc(ostream& flink, string linkname, string sourcename, string targetname,
OBJ unLinkUML)
{
        string comment, axiom, multsource, multtarget;


        //multiplicity constraints
        fskipline(flink);
        comment = "%axioms for multiplicity constraints: written only if multiplicity is
not free, i.e. different from 0 or More";
        frecordln(flink, comment);

        //recover the value for the properties multiplicity target and
        //multiplicity source of the link
        multsource = GtiNameOf(GtiGetMenu(unLinkUML, GtiSymbol("Multiplicity Source")));
        comment = "%source multiplicity is " + multsource;
        frecordln(flink, comment);
        multtarget = GtiNameOf(GtiGetMenu(unLinkUML, GtiSymbol("Multiplicity Target")));
        comment = "%target multiplicity is " + multtarget;
        frecordln(flink, comment);

        //axioms to state multiplicity constraints at the source end of the association
        if (multsource == "Just One") {
                axiom = "size(allLeftLink(" + linkname + "1, a" + targetname + "1))";
                axiom =  axiom + " = 1;";
                frecordln(flink, axiom);}
```

```
        else
                if (multsource == "Optional (0 or 1)") {
                        axiom = "~(size(allLeftLink(" + linkname + "1, a" + targetname +
")) > 1;";
                        frecordln(flink, axiom); }
                else
                        if (multsource == "1 or more") {
                                axiom = "~(size(allLeftLink(" + linkname + "1, a" +
targetname + ")) < 1;";
                                frecordln(flink, axiom);
                        };

        //axioms to state multiplicity constraints at the target end of the association

        if (multtarget == "Just One") {
                axiom = "size(allRightLink(" + linkname + "1, a" + sourcename + "1))";
                axiom =  axiom + " = 1;";
                frecordln(flink, axiom); }
        else
                if (multtarget == "Optional (0 or 1)") {
                        axiom = "~(size(allRightLink(" + linkname + "1, a" + sourcename +
")) > 1;";
                        frecordln(flink, axiom); }

                else
                        if (multtarget == "1 or more") {
                                axiom = "~(size(allRightLink(" + linkname + "1, a" +
sourcename + ")) < 1;";
                                frecordln(flink, axiom);
                                };

                if (GtiClassSymbol(GtiClassOf(unLinkUML)) == GtiSymbol("Composition
Link"))
                        rule4Composition(flink, linkname, sourcename, targetname);

}


//Rule 8 states the axioms for association identity
void rule8Assoc(ostream& flink, string linkname, string sourcename, string targetname)
{
        string comment, axiom;


        fskipline(flink);
        comment = "%axioms for association identity";
        frecordln(flink, comment);

        axiom = "identity(void(i)) = i;";
        frecordln(flink, axiom);
        axiom = "identity(addLink(" + linkname + "1" + ", " + "a" + sourcename + "1" + ",
";
        axiom = axiom + "a" + targetname + "1" + "))" + " = " + "identity" + "(" +
linkname + "1" + ");";
        frecordln(flink, axiom);
        axiom = linkname + "1" + " \\eq " + linkname + "2";
        axiom = axiom + " = " + "identity(" + linkname + "1" + ")" + " \\eq " +
"identity(" + linkname + "2" + ")";
        frecordln(flink, axiom);
}

//Rule 9 generates the sort Set for the source and target Class
//of the Association
void rule9Set(ostream& fset, string classname)
{
        string specname, comment, specvariables, specoperator, axiom;

        comment =
"%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%";
        frecordln(fset, comment);
        comment = "%specification of the type Set for " + classname + " Class ";
```

```
        frecordln(fset, comment);
        comment = "%larch file: Set" + classname + ".lp";
        frecordln(fset, comment);
        comment =
"%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%";
        frecordln(fset, comment);
        fskipline(fset);
        fskipline(fset);

        //set name section starts the formal specification for the sort
        specname = "set name Set" + classname;
        frecordln(fset, specname);
        fskipline(fset);

        //declares used sorts
        specname = "declare sorts " + classname + ", " + "Set" + classname + ", Nat";
        frecordln(fset, specname);

        //declare variables
        specvariables = "declare variables ";
        frecord(fset, specvariables);
        specvariables = "a" + classname + "1, ";
        specvariables = specvariables + "a" + classname + "2";
        specvariables = specvariables + " : " + classname + ", ";
        frecord(fset, specvariables);
        specvariables = "set" + classname + "1" + ", set" + classname + "2 : Set" +
classname;
        frecordln(fset, specvariables);

        //declare operators for Set
        specname = "declare operators";
        frecordln(fset, specname);
        specoperator = "{} :  -> Set" + classname;
        frecordln(fset, specoperator);
        specoperator = "{__} : " + classname + " -> " + "Set" + classname;
        frecordln(fset, specoperator);
        specoperator = "insert : " + classname + ", Set" + classname + " -> Set" +
classname;
        frecordln(fset, specoperator);
        specoperator = "__\\U__ : Set" + classname + ", Set" + classname + " -> Set" +
classname;
        frecordln(fset, specoperator);
        specoperator = "__\\in__ : " + classname + ", Set" + classname + " -> Bool";
        frecordln(fset, specoperator);
        specoperator = "__\\I__ : Set" + classname + ", Set" + classname + " -> Bool";
        frecordln(fset, specoperator);
        specoperator = "size : Set" + classname + " -> Nat";
        frecordln(fset, specoperator);

        //ends operators section
        endSection(fset);

        //axioms section
        axiom = "assert";
        fskipline(fset);
        frecordln(fset, axiom);
        //generators
        axiom = "sort Set" + classname + " generated by {}, insert;";
        frecordln(fset, axiom);
        fskipline(fset);

        //axioms for the operations
        axiom = "{a" + classname + "1} = insert(a" + classname + "1, {});";
        frecordln(fset, axiom);
        axiom = "~(a" + classname + "1 \\in {});";
        frecordln(fset, axiom);
        axiom = "a" + classname + "1 \\in insert(a" + classname + "2, set" + classname +
"1) <=> (a" + classname + "1";
        axiom = axiom + " \\eq a" + classname + "2 \\/ a" + classname + "1 \\in set" +
classname + "1);";
        frecordln(fset, axiom);
```

```
        axiom = "{} \\I set" + classname + "1;";
        frecordln(fset, axiom);
        axiom = "insert(a" + classname + "1, set" + classname + "1) \\I set" + classname +
"2 <=> (a" + classname + "1";
        axiom = axiom + " \\in set" + classname + "2 /\\ set" + classname + "1 \\I set" +
classname + "2);";
        frecordln(fset, axiom);
        axiom = "a" + classname + "1 \\in (set" + classname + "1 \\U set" + classname +
"2) <=> (a" + classname + "1 \\in set" + classname + "1 \\/ a";
        axiom = axiom + classname + "1 \\in set" + classname + "2);";
        frecordln(fset, axiom);

        //axioms for the size operator
        fskipline(fset);
        comment = "% axioms for size operator";
        frecordln(fset, comment);
        axiom = "size({}) = 0;";
        frecordln(fset, axiom);
        axiom = "(a" + classname + "1 \\in set" + classname + "1) => size(insert(a" +
classname + "1, set" + classname + "1)) = size(set" + classname + "1);";
        frecordln(fset, axiom);
        axiom = "~(a" + classname + "1 \\in set" + classname + "1) => size(insert(a" +
classname + "1, set" + classname + "1)) = 1+size(set" + classname + "1);";
        frecordln(fset, axiom);

        //ends axioms section
        endSection(fset);
}


//Special Rules for Compositions
string rule1Composition(string linkname)
{
        string compname;

        compname = "Comp" + linkname;
        return compname;
}


//Special Rules for Compositions
string rule2Composition(string linkname, string sourcename, string targetname)
{
        string comment;

        comment = "%specification of the composition " + linkname + " between Classes: " +
sourcename + " and " + targetname;
        return comment;
}

//Special Rules for Compositions
void rule3Composition(ostream& flink, string linkname, string sourcename, string
targetname)
{

        string  formaloperator, comment;


        fskipline(flink);
        comment = "%special operator for Composition";
        frecordln(flink, comment);

        formaloperator = "isPartOf : " + linkname + ", " + sourcename + ", " + targetname
+ "  -> Bool";
        frecordln(flink, formaloperator);
}

//Special Rules for Compositions
void rule4Composition(ostream& flink, string linkname, string sourcename, string
targetname)
{
```

```
        string  axiom, comment;


        fskipline(flink);
        comment = "%special axioms for Composition";
        frecordln(flink, comment);

        axiom = "isPartOf(addLink(" + linkname + "1, a" + sourcename + "1, a" + targetname
+ "1), a" + sourcename + "2, a" + targetname + "2)";
        axiom = axiom + " => (a" + sourcename + "1 \\eq a" + sourcename + "2)";
        axiom = axiom + " /\\ (a" + targetname + "1 \\eq a" + targetname + "2);";
        frecordln(flink, axiom);

        axiom = "~(isPartOf(void(i), a" + sourcename + "1, a" + targetname + "1);";
        frecordln(flink, axiom);

        fskipline(flink);
        axiom = "~(a" + targetname + "1 \\eq a" + targetname + "2)";
        axiom = axiom + " => (isPartOf(" + linkname + "1, a" + sourcename + "1, a" +
targetname + "1) /\\";
        axiom = axiom + "(~(isPartOf(" + linkname + "1, a" + sourcename + "1, a" +
targetname + "2))))";
        axiom = axiom + " \\/ (isPartOf(" + linkname + "1, a" + sourcename + "1, a" +
targetname + "2) /\\";
        axiom = axiom + "(~(isPartOf(" + linkname + "1, a" + sourcename + "1, a" +
targetname + "1)))));";

        frecordln(flink, axiom);
}
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%specification of the properties and behavior for the instances  of
class Library
%larch file: Library.lp
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

set name Library

declare sorts Library, IdLibrary, SetLibrary, String
declare variables aLibrary1, aLibrary2 : Library, libName1,  libName2 :
String
declare variables address1,  address2 : String, telephone1,  telephone2
: String, id : IdLibrary

declare operators
% generator
newLibrary : IdLibrary, String, String, String -> Library

% operations for identity and object equality
identity :  Library -> IdLibrary
__\eq__ :  Library, Library -> Bool

% accessors for the instance variables
getlibName :  Library -> String
setlibName :  Library, String -> Library
getaddress :  Library -> String
setaddress :  Library, String -> Library
gettelephone :  Library -> String
settelephone :  Library, String -> Library

% constants to represent examples of instances
oneLibrary :       -> Library
anotherLibrary :   -> Library

..
% axioms
assert
sort Library generated by newLibrary;
%axioms for identity
identity(newLibrary(id,libName1,address1,telephone1)) = id;

%axioms for the constants
oneLibrary= newLibrary((newIdLibrary),libName1,address1,telephone1);
anotherLibrary=
newLibrary((nextIdLibrary(newIdLibrary)),libName1,address1,telephone1);
% axiom to state the semantics for object equality
aLibrary1 \eq aLibrary2 = identity(aLibrary1) \eq identity(aLibrary2);

% axioms to state the semantics of the attribute accessors
getlibName(newLibrary(id, libName1, address1, telephone1)) = libName1;
setlibName(newLibrary(id, libName1, address1, telephone1),libName2) =
newLibrary(id, libName2, address1, telephone1);
getaddress(newLibrary(id, libName1, address1, telephone1)) = address1;
setaddress(newLibrary(id, libName1, address1, telephone1),address2) =
newLibrary(id, libName1, address2, telephone1);
```

```
gettelephone(newLibrary(id, libName1, address1, telephone1)) =
telephone1;
settelephone(newLibrary(id, libName1, address1, telephone1),telephone2)
= newLibrary(id, libName1, address1, telephone2);
..


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%specification of the properties and behavior for the instances  of
class User
%larch file: User.lp
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


set name User

declare sorts User, IdUser, SetUser, Boolean
declare variables aUser1, aUser2 : User, name1,  name2 : String
declare variables address1,  address2 : String, code1,  code2 : Nat,
active1,  active2 : Boolean, id : IdUser

declare operators
% generator
newUser : IdUser, String, String, Nat, Boolean -> User

% operations for identity and object equality
identity :  User -> IdUser
__\eq__ :  User, User -> Bool

% accessors for the instance variables
getname :  User -> String
setname :  User, String -> User
getaddress :  User -> String
setaddress :  User, String -> User
getcode :  User -> Nat
setcode :  User, Nat -> User
getactive :  User -> Boolean
setactive :  User, Boolean -> User

% constants to represent examples of instances
oneUser :    -> User
anotherUser :      -> User

..
% axioms
assert
sort User generated by newUser;
%axioms for identity
identity(newUser(id,name1,address1,code1,active1)) = id;

%axioms for the constants
oneUser= newUser((newIdUser),name1,address1,code1,active1);
anotherUser=
newUser((nextIdUser(newIdUser)),name1,address1,code1,active1);
% axiom to state the semantics for object equality
aUser1 \eq aUser2 = identity(aUser1) \eq identity(aUser2);
```

```
% axioms to state the semantics of the attribute accessors
getname(newUser(id, name1, address1, code1, active1)) = name1;
setname(newUser(id, name1, address1, code1, active1),name2) =
newUser(id, name2, address1, code1, active1);
getaddress(newUser(id, name1, address1, code1, active1)) = address1;
setaddress(newUser(id, name1, address1, code1, active1),address2) =
newUser(id, name1, address2, code1, active1);
getcode(newUser(id, name1, address1, code1, active1)) = code1;
setcode(newUser(id, name1, address1, code1, active1),code2) =
newUser(id, name1, address1, code2, active1);
getactive(newUser(id, name1, address1, code1, active1)) = active1;
setactive(newUser(id, name1, address1, code1, active1),active2) =
newUser(id, name1, address1, code1, active2);
..


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%specification of the properties and behavior for the instances  of
class Publication
%larch file: Publication.lp
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


set name Publication

declare sorts Publication, IdPublication, SetPublication, String
declare variables aPublication1, aPublication2 : Publication, title1,
title2 : String
declare variables author1,  author2 : String, publishingHouse1,
publishingHouse2 : String, id : IdPublication

declare operators
% generator
newPublication : IdPublication, String, String, String -> Publication

% operations for identity and object equality
identity :  Publication -> IdPublication
__\eq__ :  Publication, Publication -> Bool

% accessors for the instance variables
gettitle :  Publication -> String
settitle :  Publication, String -> Publication
getauthor :  Publication -> String
setauthor :  Publication, String -> Publication
getpublishingHouse :  Publication -> String
setpublishingHouse :  Publication, String -> Publication

% constants to represent examples of instances
onePublication :   -> Publication
anotherPublication :     -> Publication

..
% axioms
assert
sort Publication generated by newPublication;
%axioms for identity
identity(newPublication(id,title1,author1,publishingHouse1)) = id;
```

```
%axioms for the constants
onePublication=
newPublication((newIdPublication),title1,author1,publishingHouse1);
anotherPublication=
newPublication((nextIdPublication(newIdPublication)),title1,author1,pub
lishingHouse1);
% axiom to state the semantics for object equality
aPublication1 \eq aPublication2 = identity(aPublication1) \eq
identity(aPublication2);

% axioms to state the semantics of the attribute accessors
gettitle(newPublication(id, title1, author1, publishingHouse1)) =
title1;
settitle(newPublication(id, title1, author1, publishingHouse1),title2)
= newPublication(id, title2, author1, publishingHouse1);
getauthor(newPublication(id, title1, author1, publishingHouse1)) =
author1;
setauthor(newPublication(id, title1, author1,
publishingHouse1),author2) = newPublication(id, title1, author2,
publishingHouse1);
getpublishingHouse(newPublication(id, title1, author1,
publishingHouse1)) = publishingHouse1;
setpublishingHouse(newPublication(id, title1, author1,
publishingHouse1),publishingHouse2) = newPublication(id, title1,
author1, publishingHouse2);
..

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%specification of the properties and behavior for the instances  of
class Copy
%larch file: Copy.lp
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


set name Copy

declare sorts Copy, IdCopy, SetCopy, Nat
declare variables aCopy1, aCopy2 : Copy, copyNumber1,  copyNumber2 :
Nat
declare variables , id : IdCopy

declare operators
% generator
newCopy : IdCopy, Nat -> Copy

% operations for identity and object equality
identity :  Copy -> IdCopy
__\eq__ :  Copy, Copy -> Bool

% accessors for the instance variables
getcopyNumber :  Copy -> Nat
setcopyNumber :  Copy, Nat -> Copy

% constants to represent examples of instances
oneCopy :    -> Copy
anotherCopy :      -> Copy
```

```
..
% axioms
assert
sort Copy generated by newCopy;
%axioms for identity
identity(newCopy(id,copyNumber1)) = id;

%axioms for the constants
oneCopy= newCopy((newIdCopy),copyNumber1);
anotherCopy= newCopy((nextIdCopy(newIdCopy)),copyNumber1);
% axiom to state the semantics for object equality
aCopy1 \eq aCopy2 = identity(aCopy1) \eq identity(aCopy2);

% axioms to state the semantics of the attribute accessors
getcopyNumber(newCopy(id, copyNumber1)) = copyNumber1;
setcopyNumber(newCopy(id, copyNumber1),copyNumber2) = newCopy(id,
copyNumber2);
..

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%specification of the properties and behavior for the instances  of
class Loan
%larch file: Loan.lp
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


set name Loan

declare sorts Loan, IdLoan, SetLoan, Nat
declare variables aLoan1, aLoan2 : Loan, numberLoan1,  numberLoan2 :
Nat
declare variables situation1,  situation2 : Boolean, duration1,
duration2 : Nat, id : IdLoan

declare operators
% generator
newLoan : IdLoan, Nat, Boolean, Nat -> Loan

% operations for identity and object equality
identity :  Loan -> IdLoan
__\eq__ :  Loan, Loan -> Bool

% accessors for the instance variables
getnumberLoan :  Loan -> Nat
setnumberLoan :  Loan, Nat -> Loan
getsituation :  Loan -> Boolean
setsituation :  Loan, Boolean -> Loan
getduration :  Loan -> Nat
setduration :  Loan, Nat -> Loan

% constants to represent examples of instances
oneLoan :    -> Loan
anotherLoan :      -> Loan

..
% axioms
```

```
assert
sort Loan generated by newLoan;
%axioms for identity
identity(newLoan(id,numberLoan1,situation1,duration1)) = id;

%axioms for the constants
oneLoan= newLoan((newIdLoan),numberLoan1,situation1,duration1);
anotherLoan=
newLoan((nextIdLoan(newIdLoan)),numberLoan1,situation1,duration1);
% axiom to state the semantics for object equality
aLoan1 \eq aLoan2 = identity(aLoan1) \eq identity(aLoan2);

% axioms to state the semantics of the attribute accessors
getnumberLoan(newLoan(id, numberLoan1, situation1, duration1)) =
numberLoan1;
setnumberLoan(newLoan(id, numberLoan1, situation1,
duration1),numberLoan2) = newLoan(id, numberLoan2, situation1,
duration1);
getsituation(newLoan(id, numberLoan1, situation1, duration1)) =
situation1;
setsituation(newLoan(id, numberLoan1, situation1,
duration1),situation2) = newLoan(id, numberLoan1, situation2,
duration1);
getduration(newLoan(id, numberLoan1, situation1, duration1)) =
duration1;
setduration(newLoan(id, numberLoan1, situation1, duration1),duration2)
= newLoan(id, numberLoan1, situation1, duration2);
..

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%specification of the properties and behavior for the instances  of
class LocalUse
%larch file: LocalUse.lp
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


set name LocalUse

declare sorts LocalUse, IdLocalUse, SetLocalUse, Nat
declare variables aLocalUse1, aLocalUse2 : LocalUse, hours1,  hours2 :
Nat
declare variables , id : IdLocalUse

declare operators
% generator
newLocalUse : IdLocalUse, Nat -> LocalUse

% operations for identity and object equality
identity :  LocalUse -> IdLocalUse
__\eq__ :  LocalUse, LocalUse -> Bool

% accessors for the instance variables
gethours :  LocalUse -> Nat
sethours :  LocalUse, Nat -> LocalUse

% constants to represent examples of instances
oneLocalUse :      -> LocalUse
```

```
anotherLocalUse :  -> LocalUse

..

% axioms
assert
sort LocalUse generated by newLocalUse;
%axioms for identity
identity(newLocalUse(id,hours1)) = id;

%axioms for the constants
oneLocalUse= newLocalUse((newIdLocalUse),hours1);
anotherLocalUse= newLocalUse((nextIdLocalUse(newIdLocalUse)),hours1);
% axiom to state the semantics for object equality
aLocalUse1 \eq aLocalUse2 = identity(aLocalUse1) \eq
identity(aLocalUse2);

% axioms to state the semantics of the attribute accessors
gethours(newLocalUse(id, hours1)) = hours1;
sethours(newLocalUse(id, hours1),hours2) = newLocalUse(id, hours2);
..


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%specification of the association LibUse between Classes: Loan and User
%larch file: LibUse.lp
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


set name LibUse

declare sorts LibUse, Loan, User, Nat, SetLoan, SetUser
declare variables LibUse1, LibUse2 : LibUse, aLoan1, aLoan2 : Loan,
aUser1, aUser2 : User
declare variables SetLoan1, SetLoan2 : SetLoan, SetUser1, SetUser2 :
SetUser, i : Nat

declare operators
%generator of an empty association
void :      Nat   -> LibUse

%association identity
identity :  LibUse  -> Nat
__\eq__  :  LibUse, LibUse  -> Bool

%operators to create and remove links
addLink : LibUse, Loan, User -> LibUse
removeLink : LibUse, Loan, User -> LibUse

%operator to test if two instances are linked
isLinked : LibUse, Loan, User -> Bool

%operator to test if one instance is linked
isLeftLinked : LibUse, User -> Bool
isRightLinked : LibUse, Loan -> Bool

%operator to test if the association is Empty
```

```
isEmpty : LibUse -> Bool

%operator to return the Set of instances linked to a given instance
allLeftLink : LibUse, User -> SetLoan
allRightLink : LibUse, Loan -> SetUser

%operators for multiplicity values
leftMultiplicity : LibUse, User -> Nat
rightMultiplicity : LibUse, Loan  -> Nat
..

% axioms
assert
sort LibUse generated by void, addLink;

%axiom to state that tuples of instance values cannot be equal in an
association
(aLoan1 \eq aLoan2) /\ (aUser1 \eq aUser2) => addLink(addLink(LibUse1,
aLoan1, aUser1), aLoan2, aUser2) = addLink(LibUse1, aLoan1, aUser1);

%axioms for the isEmpty operation
isEmpty(void(i));
~(isEmpty(addLink(LibUse1, aLoan1, aUser1)));

%state when two instances of object types are linked
~(isLinked(void(i),aLoan1, aUser1));
isLinked(addLink(LibUse1, aLoan1, aUser1),aLoan2, aUser2) = ((aLoan1
\eq aLoan2 /\ aUser1 \eq aUser2) \/ isLinked(LibUse1, aLoan2, aUser2));

%state the semantics for the operations allLeftLink and allRightLink
through addLink generator
(aUser1 \eq aUser2) => allLeftLink(addLink(LibUse1, aLoan1, aUser2),
aUser1) = insert(aLoan1, allLeftLink(LibUse1, aUser1));
(aLoan1 \eq aLoan2) => allRightLink(addLink(LibUse1, aLoan2, aUser1),
aLoan1) = insert(aUser1, allRightLink(LibUse1, aLoan1));
~(aUser1 \eq aUser2) => allLeftLink(addLink(LibUse1, aLoan1, aUser2),
aUser1) = allLeftLink(LibUse1, aUser1);
~(aLoan1 \eq aLoan2) => allRightLink(addLink(LibUse1, aLoan2, aUser1),
aLoan1) = allRightLink(LibUse1, aLoan1);

%state the semantics for allLeftLink and allRightLink through void
generator
allLeftLink(void(i), aUser1) = {}: SetUser;
allRightLink(void(i), aLoan1) = {}: SetLoan;

%state when one instance is linked through the association
~(isLeftLinked(void(i), aUser1));
~(isRightLinked(void(i), aLoan1));
isLeftLinked(addLink(LibUse1, aLoan1, aUser1), aUser2) = ((aUser1 \eq
aUser2) \/ isLeftLinked(LibUse1, aUser2));
isRightLinked(addLink(LibUse1, aLoan1, aUser1), aLoan2) = ((aLoan1 \eq
aLoan2) \/ isRightLinked(LibUse1, aLoan2));

%axioms for left and rightMultiplicity operators
leftMultiplicity(void(i), aUser1) = 0;
rightMultiplicity(void(i), aLoan1) = 0;
```

```
(aUser1 \eq aUser2) => leftMultiplicity(addLink(LibUse1, aLoan2,
aUser2),aUser1) = leftMultiplicity( LibUse1, aUser1) + 1;
~(aUser1 \eq aUser2) => leftMultiplicity(addLink(LibUse1, aLoan2,
aUser2),aUser1) = leftMultiplicity( LibUse1, aUser1);

(aLoan1 \eq aLoan2) => rightMultiplicity(addLink(LibUse1, aLoan2,
aUser2),aLoan1) = rightMultiplicity( LibUse1, aLoan1) + 1;
~(aLoan1 \eq aLoan2) => rightMultiplicity(addLink(LibUse1, aLoan2,
aUser2),aLoan1) = rightMultiplicity( LibUse1, aLoan1);

%axioms for multiplicity constraints: written only if multiplicity is
not free, i.e. different from 0 or More
%source multiplicity is Just One
%target multiplicity is Many (0 or plus)
size(allLeftLink(LibUse1, aUser1)) = 1;

%axioms for association identity
identity(void(i)) = i;
identity(addLink(LibUse1, aLoan1, aUser1)) = identity(LibUse1);
LibUse1 \eq LibUse2 = identity(LibUse1) \eq identity(LibUse2)
..

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%specification of the association LoaUse between Classes: Loan and User
%larch file: LoaUse.lp
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


set name LoaUse

declare sorts LoaUse, Loan, User, Nat, SetLoan, SetUser
declare variables LoaUse1, LoaUse2 : LoaUse, aLoan1, aLoan2 : Loan,
aUser1, aUser2 : User
declare variables SetLoan1, SetLoan2 : SetLoan, SetUser1, SetUser2 :
SetUser, i : Nat

declare operators
%generator of an empty association
void :      Nat   -> LoaUse

%association identity
identity :  LoaUse  -> Nat
__\eq__  :  LoaUse, LoaUse  -> Bool

%operators to create and remove links
addLink : LoaUse, Loan, User -> LoaUse
removeLink : LoaUse, Loan, User -> LoaUse

%operator to test if two instances are linked
isLinked : LoaUse, Loan, User -> Bool

%operator to test if one instance is linked
isLeftLinked : LoaUse, User -> Bool
isRightLinked : LoaUse, Loan -> Bool

%operator to test if the association is Empty
isEmpty : LoaUse -> Bool
```

```
%operator to return the Set of instances linked to a given instance
allLeftLink : LoaUse, User -> SetLoan
allRightLink : LoaUse, Loan -> SetUser

%operators for multiplicity values
leftMultiplicity : LoaUse, User -> Nat
rightMultiplicity : LoaUse, Loan  -> Nat
..

% axioms
assert
sort LoaUse generated by void, addLink;

%axiom to state that tuples of instance values cannot be equal in an
association
(aLoan1 \eq aLoan2) /\ (aUser1 \eq aUser2) => addLink(addLink(LoaUse1,
aLoan1, aUser1), aLoan2, aUser2) = addLink(LoaUse1, aLoan1, aUser1);

%axioms for the isEmpty operation
isEmpty(void(i));
~(isEmpty(addLink(LoaUse1, aLoan1, aUser1)));

%state when two instances of object types are linked
~(isLinked(void(i),aLoan1, aUser1));
isLinked(addLink(LoaUse1, aLoan1, aUser1),aLoan2, aUser2) = ((aLoan1
\eq aLoan2 /\ aUser1 \eq aUser2) \/ isLinked(LoaUse1, aLoan2, aUser2));

%state the semantics for the operations allLeftLink and allRightLink
through addLink generator
(aUser1 \eq aUser2) => allLeftLink(addLink(LoaUse1, aLoan1, aUser2),
aUser1) = insert(aLoan1, allLeftLink(LoaUse1, aUser1));
(aLoan1 \eq aLoan2) => allRightLink(addLink(LoaUse1, aLoan2, aUser1),
aLoan1) = insert(aUser1, allRightLink(LoaUse1, aLoan1));
~(aUser1 \eq aUser2) => allLeftLink(addLink(LoaUse1, aLoan1, aUser2),
aUser1) = allLeftLink(LoaUse1, aUser1);
~(aLoan1 \eq aLoan2) => allRightLink(addLink(LoaUse1, aLoan2, aUser1),
aLoan1) = allRightLink(LoaUse1, aLoan1);

%state the semantics for allLeftLink and allRightLink through void
generator
allLeftLink(void(i), aUser1) = {}: SetUser;
allRightLink(void(i), aLoan1) = {}: SetLoan;

%state when one instance is linked through the association
~(isLeftLinked(void(i), aUser1));
~(isRightLinked(void(i), aLoan1));
isLeftLinked(addLink(LoaUse1, aLoan1, aUser1), aUser2) = ((aUser1 \eq
aUser2) \/ isLeftLinked(LoaUse1, aUser2));
isRightLinked(addLink(LoaUse1, aLoan1, aUser1), aLoan2) = ((aLoan1 \eq
aLoan2) \/ isRightLinked(LoaUse1, aLoan2));

%axioms for left and rightMultiplicity operators
leftMultiplicity(void(i), aUser1) = 0;
rightMultiplicity(void(i), aLoan1) = 0;
(aUser1 \eq aUser2) => leftMultiplicity(addLink(LoaUse1, aLoan2,
aUser2),aUser1) = leftMultiplicity( LoaUse1, aUser1) + 1;
```

```
~(aUser1 \eq aUser2) => leftMultiplicity(addLink(LoaUse1, aLoan2,
aUser2),aUser1) = leftMultiplicity( LoaUse1, aUser1);

(aLoan1 \eq aLoan2) => rightMultiplicity(addLink(LoaUse1, aLoan2,
aUser2),aLoan1) = rightMultiplicity( LoaUse1, aLoan1) + 1;
~(aLoan1 \eq aLoan2) => rightMultiplicity(addLink(LoaUse1, aLoan2,
aUser2),aLoan1) = rightMultiplicity( LoaUse1, aLoan1);

%axioms for multiplicity constraints: written only if multiplicity is
not free, i.e. different from 0 or More
%source multiplicity is Many (0 or plus)
%target multiplicity is Just One
size(allRightLink(LoaUse1, aLoan1)) = 1;

%axioms for association identity
identity(void(i)) = i;
identity(addLink(LoaUse1, aLoan1, aUser1)) = identity(LoaUse1);
LoaUse1 \eq LoaUse2 = identity(LoaUse1) \eq identity(LoaUse2)
..

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%specification of the association LocUse between Classes: Loan and User
%larch file: LocUse.lp
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


set name LocUse

declare sorts LocUse, Loan, User, Nat, SetLoan, SetUser
declare variables LocUse1, LocUse2 : LocUse, aLoan1, aLoan2 : Loan,
aUser1, aUser2 : User
declare variables SetLoan1, SetLoan2 : SetLoan, SetUser1, SetUser2 :
SetUser, i : Nat

declare operators
%generator of an empty association
void :      Nat   -> LocUse

%association identity
identity :  LocUse  -> Nat
__\eq__  :  LocUse, LocUse  -> Bool

%operators to create and remove links
addLink : LocUse, Loan, User -> LocUse
removeLink : LocUse, Loan, User -> LocUse

%operator to test if two instances are linked
isLinked : LocUse, Loan, User -> Bool

%operator to test if one instance is linked
isLeftLinked : LocUse, User -> Bool
isRightLinked : LocUse, Loan -> Bool

%operator to test if the association is Empty
isEmpty : LocUse -> Bool
```

```
%operator to return the Set of instances linked to a given instance
allLeftLink : LocUse, User -> SetLoan
allRightLink : LocUse, Loan -> SetUser

%operators for multiplicity values
leftMultiplicity : LocUse, User -> Nat
rightMultiplicity : LocUse, Loan  -> Nat
..

% axioms
assert
sort LocUse generated by void, addLink;

%axiom to state that tuples of instance values cannot be equal in an
association
(aLoan1 \eq aLoan2) /\ (aUser1 \eq aUser2) => addLink(addLink(LocUse1,
aLoan1, aUser1), aLoan2, aUser2) = addLink(LocUse1, aLoan1, aUser1);

%axioms for the isEmpty operation
isEmpty(void(i));
~(isEmpty(addLink(LocUse1, aLoan1, aUser1)));

%state when two instances of object types are linked
~(isLinked(void(i),aLoan1, aUser1));
isLinked(addLink(LocUse1, aLoan1, aUser1),aLoan2, aUser2) = ((aLoan1
\eq aLoan2 /\ aUser1 \eq aUser2) \/ isLinked(LocUse1, aLoan2, aUser2));

%state the semantics for the operations allLeftLink and allRightLink
through addLink generator
(aUser1 \eq aUser2) => allLeftLink(addLink(LocUse1, aLoan1, aUser2),
aUser1) = insert(aLoan1, allLeftLink(LocUse1, aUser1));
(aLoan1 \eq aLoan2) => allRightLink(addLink(LocUse1, aLoan2, aUser1),
aLoan1) = insert(aUser1, allRightLink(LocUse1, aLoan1));
~(aUser1 \eq aUser2) => allLeftLink(addLink(LocUse1, aLoan1, aUser2),
aUser1) = allLeftLink(LocUse1, aUser1);
~(aLoan1 \eq aLoan2) => allRightLink(addLink(LocUse1, aLoan2, aUser1),
aLoan1) = allRightLink(LocUse1, aLoan1);

%state the semantics for allLeftLink and allRightLink through void
generator
allLeftLink(void(i), aUser1) = {}: SetUser;
allRightLink(void(i), aLoan1) = {}: SetLoan;

%state when one instance is linked through the association
~(isLeftLinked(void(i), aUser1));
~(isRightLinked(void(i), aLoan1));
isLeftLinked(addLink(LocUse1, aLoan1, aUser1), aUser2) = ((aUser1 \eq
aUser2) \/ isLeftLinked(LocUse1, aUser2));
isRightLinked(addLink(LocUse1, aLoan1, aUser1), aLoan2) = ((aLoan1 \eq
aLoan2) \/ isRightLinked(LocUse1, aLoan2));

%axioms for left and rightMultiplicity operators
leftMultiplicity(void(i), aUser1) = 0;
rightMultiplicity(void(i), aLoan1) = 0;
(aUser1 \eq aUser2) => leftMultiplicity(addLink(LocUse1, aLoan2,
aUser2),aUser1) = leftMultiplicity( LocUse1, aUser1) + 1;
```

```
~(aUser1 \eq aUser2) => leftMultiplicity(addLink(LocUse1, aLoan2,
aUser2),aUser1) = leftMultiplicity( LocUse1, aUser1);

(aLoan1 \eq aLoan2) => rightMultiplicity(addLink(LocUse1, aLoan2,
aUser2),aLoan1) = rightMultiplicity( LocUse1, aLoan1) + 1;
~(aLoan1 \eq aLoan2) => rightMultiplicity(addLink(LocUse1, aLoan2,
aUser2),aLoan1) = rightMultiplicity( LocUse1, aLoan1);

%axioms for multiplicity constraints: written only if multiplicity is
not free, i.e. different from 0 or More
%source multiplicity is Many (0 or plus)
%target multiplicity is Just One
size(allRightLink(LocUse1, aLoan1)) = 1;

%axioms for association identity
identity(void(i)) = i;
identity(addLink(LocUse1, aLoan1, aUser1)) = identity(LocUse1);
LocUse1 \eq LocUse2 = identity(LocUse1) \eq identity(LocUse2)
..

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%specification of the association PubCop between Classes: Publication
and Copy
%larch file: PubCop.lp
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


set name PubCop

declare sorts PubCop, Publication, Copy, Nat, SetPublication, SetCopy
declare variables PubCop1, PubCop2 : PubCop, aPublication1,
aPublication2 : Publication, aCopy1, aCopy2 : Copy
declare variables SetPublication1, SetPublication2 : SetPublication,
SetCopy1, SetCopy2 : SetCopy, i : Nat

declare operators
%generator of an empty association
void :      Nat   -> PubCop

%association identity
identity :  PubCop  -> Nat
__\eq__  :  PubCop, PubCop  -> Bool

%operators to create and remove links
addLink : PubCop, Publication, Copy -> PubCop
removeLink : PubCop, Publication, Copy -> PubCop

%operator to test if two instances are linked
isLinked : PubCop, Publication, Copy -> Bool

%operator to test if one instance is linked
isLeftLinked : PubCop, Copy -> Bool
isRightLinked : PubCop, Publication -> Bool

%operator to test if the association is Empty
isEmpty : PubCop -> Bool
```

```
%operator to return the Set of instances linked to a given instance
allLeftLink : PubCop, Copy -> SetPublication
allRightLink : PubCop, Publication -> SetCopy

%operators for multiplicity values
leftMultiplicity : PubCop, Copy -> Nat
rightMultiplicity : PubCop, Publication  -> Nat
..

% axioms
assert
sort PubCop generated by void, addLink;

%axiom to state that tuples of instance values cannot be equal in an
association
(aPublication1 \eq aPublication2) /\ (aCopy1 \eq aCopy2) =>
addLink(addLink(PubCop1, aPublication1, aCopy1), aPublication2, aCopy2)
= addLink(PubCop1, aPublication1, aCopy1);

%axioms for the isEmpty operation
isEmpty(void(i));
~(isEmpty(addLink(PubCop1, aPublication1, aCopy1)));

%state when two instances of object types are linked
~(isLinked(void(i),aPublication1, aCopy1));
isLinked(addLink(PubCop1, aPublication1, aCopy1),aPublication2, aCopy2)
= ((aPublication1 \eq aPublication2 /\ aCopy1 \eq aCopy2) \/
isLinked(PubCop1, aPublication2, aCopy2));

%state the semantics for the operations allLeftLink and allRightLink
through addLink generator
(aCopy1 \eq aCopy2) => allLeftLink(addLink(PubCop1, aPublication1,
aCopy2), aCopy1) = insert(aPublication1, allLeftLink(PubCop1, aCopy1));
(aPublication1 \eq aPublication2) => allRightLink(addLink(PubCop1,
aPublication2, aCopy1), aPublication1) = insert(aCopy1,
allRightLink(PubCop1, aPublication1));
~(aCopy1 \eq aCopy2) => allLeftLink(addLink(PubCop1, aPublication1,
aCopy2), aCopy1) = allLeftLink(PubCop1, aCopy1);
~(aPublication1 \eq aPublication2) => allRightLink(addLink(PubCop1,
aPublication2, aCopy1), aPublication1) = allRightLink(PubCop1,
aPublication1);

%state the semantics for allLeftLink and allRightLink through void
generator
allLeftLink(void(i), aCopy1) = {}: SetCopy;
allRightLink(void(i), aPublication1) = {}: SetPublication;

%state when one instance is linked through the association
~(isLeftLinked(void(i), aCopy1));
~(isRightLinked(void(i), aPublication1));
isLeftLinked(addLink(PubCop1, aPublication1, aCopy1), aCopy2) =
((aCopy1 \eq aCopy2) \/ isLeftLinked(PubCop1, aCopy2));
isRightLinked(addLink(PubCop1, aPublication1, aCopy1), aPublication2) =
((aPublication1 \eq aPublication2) \/ isRightLinked(PubCop1,
aPublication2));
```

```
%axioms for left and rightMultiplicity operators
leftMultiplicity(void(i), aCopy1) = 0;
rightMultiplicity(void(i), aPublication1) = 0;
(aCopy1 \eq aCopy2) => leftMultiplicity(addLink(PubCop1, aPublication2,
aCopy2),aCopy1) = leftMultiplicity( PubCop1, aCopy1) + 1;
~(aCopy1 \eq aCopy2) => leftMultiplicity(addLink(PubCop1,
aPublication2, aCopy2),aCopy1) = leftMultiplicity( PubCop1, aCopy1);

(aPublication1 \eq aPublication2) => rightMultiplicity(addLink(PubCop1,
aPublication2, aCopy2),aPublication1) = rightMultiplicity( PubCop1,
aPublication1) + 1;
~(aPublication1 \eq aPublication2) =>
rightMultiplicity(addLink(PubCop1, aPublication2,
aCopy2),aPublication1) = rightMultiplicity( PubCop1, aPublication1);

%axioms for multiplicity constraints: written only if multiplicity is
not free, i.e. different from 0 or More
%source multiplicity is Just One
%target multiplicity is 1 or more
size(allLeftLink(PubCop1, aCopy1)) = 1;
~(size(allRightLink(PubCop1, aPublication)) < 1;

%axioms for association identity
identity(void(i)) = i;
identity(addLink(PubCop1, aPublication1, aCopy1)) = identity(PubCop1);
PubCop1 \eq PubCop2 = identity(PubCop1) \eq identity(PubCop2)
..

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%specification of the association PubLib between Classes: Publication
and Library
%larch file: PubLib.lp
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


set name PubLib

declare sorts PubLib, Publication, Library, Nat, SetPublication,
SetLibrary
declare variables PubLib1, PubLib2 : PubLib, aPublication1,
aPublication2 : Publication, aLibrary1, aLibrary2 : Library
declare variables SetPublication1, SetPublication2 : SetPublication,
SetLibrary1, SetLibrary2 : SetLibrary, i : Nat

declare operators
%generator of an empty association
void :      Nat   -> PubLib

%association identity
identity :  PubLib  -> Nat
__\eq__  :  PubLib, PubLib  -> Bool

%operators to create and remove links
addLink : PubLib, Publication, Library -> PubLib
removeLink : PubLib, Publication, Library -> PubLib
```

```
%operator to test if two instances are linked
isLinked : PubLib, Publication, Library -> Bool

%operator to test if one instance is linked
isLeftLinked : PubLib, Library -> Bool
isRightLinked : PubLib, Publication -> Bool

%operator to test if the association is Empty
isEmpty : PubLib -> Bool

%operator to return the Set of instances linked to a given instance
allLeftLink : PubLib, Library -> SetPublication
allRightLink : PubLib, Publication -> SetLibrary

%operators for multiplicity values
leftMultiplicity : PubLib, Library -> Nat
rightMultiplicity : PubLib, Publication  -> Nat
..

% axioms
assert
sort PubLib generated by void, addLink;

%axiom to state that tuples of instance values cannot be equal in an
association
(aPublication1 \eq aPublication2) /\ (aLibrary1 \eq aLibrary2) =>
addLink(addLink(PubLib1, aPublication1, aLibrary1), aPublication2,
aLibrary2) = addLink(PubLib1, aPublication1, aLibrary1);

%axioms for the isEmpty operation
isEmpty(void(i));
~(isEmpty(addLink(PubLib1, aPublication1, aLibrary1)));

%state when two instances of object types are linked
~(isLinked(void(i),aPublication1, aLibrary1));
isLinked(addLink(PubLib1, aPublication1, aLibrary1),aPublication2,
aLibrary2) = ((aPublication1 \eq aPublication2 /\ aLibrary1 \eq
aLibrary2) \/ isLinked(PubLib1, aPublication2, aLibrary2));

%state the semantics for the operations allLeftLink and allRightLink
through addLink generator
(aLibrary1 \eq aLibrary2) => allLeftLink(addLink(PubLib1,
aPublication1, aLibrary2), aLibrary1) = insert(aPublication1,
allLeftLink(PubLib1, aLibrary1));
(aPublication1 \eq aPublication2) => allRightLink(addLink(PubLib1,
aPublication2, aLibrary1), aPublication1) = insert(aLibrary1,
allRightLink(PubLib1, aPublication1));
~(aLibrary1 \eq aLibrary2) => allLeftLink(addLink(PubLib1,
aPublication1, aLibrary2), aLibrary1) = allLeftLink(PubLib1,
aLibrary1);
~(aPublication1 \eq aPublication2) => allRightLink(addLink(PubLib1,
aPublication2, aLibrary1), aPublication1) = allRightLink(PubLib1,
aPublication1);

%state the semantics for allLeftLink and allRightLink through void
generator
allLeftLink(void(i), aLibrary1) = {}: SetLibrary;
```

```
allRightLink(void(i), aPublication1) = {}: SetPublication;

%state when one instance is linked through the association
~(isLeftLinked(void(i), aLibrary1));
~(isRightLinked(void(i), aPublication1));
isLeftLinked(addLink(PubLib1, aPublication1, aLibrary1), aLibrary2) =
((aLibrary1 \eq aLibrary2) \/ isLeftLinked(PubLib1, aLibrary2));
isRightLinked(addLink(PubLib1, aPublication1, aLibrary1),
aPublication2) = ((aPublication1 \eq aPublication2) \/
isRightLinked(PubLib1, aPublication2));

%axioms for left and rightMultiplicity operators
leftMultiplicity(void(i), aLibrary1) = 0;
rightMultiplicity(void(i), aPublication1) = 0;
(aLibrary1 \eq aLibrary2) => leftMultiplicity(addLink(PubLib1,
aPublication2, aLibrary2),aLibrary1) = leftMultiplicity( PubLib1,
aLibrary1) + 1;
~(aLibrary1 \eq aLibrary2) => leftMultiplicity(addLink(PubLib1,
aPublication2, aLibrary2),aLibrary1) = leftMultiplicity( PubLib1,
aLibrary1);

(aPublication1 \eq aPublication2) => rightMultiplicity(addLink(PubLib1,
aPublication2, aLibrary2),aPublication1) = rightMultiplicity( PubLib1,
aPublication1) + 1;
~(aPublication1 \eq aPublication2) =>
rightMultiplicity(addLink(PubLib1, aPublication2,
aLibrary2),aPublication1) = rightMultiplicity( PubLib1, aPublication1);

%axioms for multiplicity constraints: written only if multiplicity is
not free, i.e. different from 0 or More
%source multiplicity is Many (0 or plus)
%target multiplicity is Just One
size(allRightLink(PubLib1, aPublication1)) = 1;

%axioms for association identity
identity(void(i)) = i;
identity(addLink(PubLib1, aPublication1, aLibrary1)) =
identity(PubLib1);
PubLib1 \eq PubLib2 = identity(PubLib1) \eq identity(PubLib2)
..

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%specification of the composition CompCopLoa between Classes: Copy and
Loan
%larch file: CompCopLoa.lp
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


set name CompCopLoa

declare sorts CompCopLoa, Copy, Loan, Nat, SetCopy, SetLoan
declare variables CompCopLoa1, CompCopLoa2 : CompCopLoa, aCopy1, aCopy2
: Copy, aLoan1, aLoan2 : Loan
declare variables SetCopy1, SetCopy2 : SetCopy, SetLoan1, SetLoan2 :
SetLoan, i : Nat
```

```
declare operators
%generator of an empty association
void :      Nat   -> CompCopLoa

%association identity
identity :  CompCopLoa  -> Nat
__\eq__  :  CompCopLoa, CompCopLoa  -> Bool

%operators to create and remove links
addLink : CompCopLoa, Copy, Loan -> CompCopLoa
removeLink : CompCopLoa, Copy, Loan -> CompCopLoa

%operator to test if two instances are linked
isLinked : CompCopLoa, Copy, Loan -> Bool

%operator to test if one instance is linked
isLeftLinked : CompCopLoa, Loan -> Bool
isRightLinked : CompCopLoa, Copy -> Bool

%operator to test if the association is Empty
isEmpty : CompCopLoa -> Bool

%operator to return the Set of instances linked to a given instance
allLeftLink : CompCopLoa, Loan -> SetCopy
allRightLink : CompCopLoa, Copy -> SetLoan

%operators for multiplicity values
leftMultiplicity : CompCopLoa, Loan -> Nat
rightMultiplicity : CompCopLoa, Copy  -> Nat

%special operator for Composition
isPartOf : CompCopLoa, Copy, Loan  -> Bool
..

% axioms
assert
sort CompCopLoa generated by void, addLink;

%axiom to state that tuples of instance values cannot be equal in an
association
(aCopy1 \eq aCopy2) /\ (aLoan1 \eq aLoan2) =>
addLink(addLink(CompCopLoa1, aCopy1, aLoan1), aCopy2, aLoan2) =
addLink(CompCopLoa1, aCopy1, aLoan1);

%axioms for the isEmpty operation
isEmpty(void(i));
~(isEmpty(addLink(CompCopLoa1, aCopy1, aLoan1)));

%state when two instances of object types are linked
~(isLinked(void(i),aCopy1, aLoan1));
isLinked(addLink(CompCopLoa1, aCopy1, aLoan1),aCopy2, aLoan2) =
((aCopy1 \eq aCopy2 /\ aLoan1 \eq aLoan2) \/ isLinked(CompCopLoa1,
aCopy2, aLoan2));

%state the semantics for the operations allLeftLink and allRightLink
through addLink generator
```

```
(aLoan1 \eq aLoan2) => allLeftLink(addLink(CompCopLoa1, aCopy1,
aLoan2), aLoan1) = insert(aCopy1, allLeftLink(CompCopLoa1, aLoan1));
(aCopy1 \eq aCopy2) => allRightLink(addLink(CompCopLoa1, aCopy2,
aLoan1), aCopy1) = insert(aLoan1, allRightLink(CompCopLoa1, aCopy1));
~(aLoan1 \eq aLoan2) => allLeftLink(addLink(CompCopLoa1, aCopy1,
aLoan2), aLoan1) = allLeftLink(CompCopLoa1, aLoan1);
~(aCopy1 \eq aCopy2) => allRightLink(addLink(CompCopLoa1, aCopy2,
aLoan1), aCopy1) = allRightLink(CompCopLoa1, aCopy1);

%state the semantics for allLeftLink and allRightLink through void
generator
allLeftLink(void(i), aLoan1) = {}: SetLoan;
allRightLink(void(i), aCopy1) = {}: SetCopy;

%state when one instance is linked through the association
~(isLeftLinked(void(i), aLoan1));
~(isRightLinked(void(i), aCopy1));
isLeftLinked(addLink(CompCopLoa1, aCopy1, aLoan1), aLoan2) = ((aLoan1
\eq aLoan2) \/ isLeftLinked(CompCopLoa1, aLoan2));
isRightLinked(addLink(CompCopLoa1, aCopy1, aLoan1), aCopy2) = ((aCopy1
\eq aCopy2) \/ isRightLinked(CompCopLoa1, aCopy2));

%axioms for left and rightMultiplicity operators
leftMultiplicity(void(i), aLoan1) = 0;
rightMultiplicity(void(i), aCopy1) = 0;
(aLoan1 \eq aLoan2) => leftMultiplicity(addLink(CompCopLoa1, aCopy2,
aLoan2),aLoan1) = leftMultiplicity( CompCopLoa1, aLoan1) + 1;
~(aLoan1 \eq aLoan2) => leftMultiplicity(addLink(CompCopLoa1, aCopy2,
aLoan2),aLoan1) = leftMultiplicity( CompCopLoa1, aLoan1);

(aCopy1 \eq aCopy2) => rightMultiplicity(addLink(CompCopLoa1, aCopy2,
aLoan2),aCopy1) = rightMultiplicity( CompCopLoa1, aCopy1) + 1;
~(aCopy1 \eq aCopy2) => rightMultiplicity(addLink(CompCopLoa1, aCopy2,
aLoan2),aCopy1) = rightMultiplicity( CompCopLoa1, aCopy1);

%axioms for multiplicity constraints: written only if multiplicity is
not free, i.e. different from 0 or More
%source multiplicity is 1 or more
%target multiplicity is Just One
~(size(allLeftLink(CompCopLoa1, aLoan)) < 1;
size(allRightLink(CompCopLoa1, aCopy1)) = 1;

%special axioms for Composition
isPartOf(addLink(CompCopLoa1, aCopy1, aLoan1), aCopy2, aLoan2) =>
(aCopy1 \eq aCopy2) /\ (aLoan1 \eq aLoan2);
~(isPartOf(void(i), aCopy1, aLoan1);

~(aLoan1 \eq aLoan2) => (isPartOf(CompCopLoa1, aCopy1, aLoan1)
/\(~(isPartOf(CompCopLoa1, aCopy1, aLoan2)))) \/ (isPartOf(CompCopLoa1,
aCopy1, aLoan2) /\(~(isPartOf(CompCopLoa1, aCopy1, aLoan1))));

%axioms for association identity
identity(void(i)) = i;
identity(addLink(CompCopLoa1, aCopy1, aLoan1)) = identity(CompCopLoa1);
CompCopLoa1 \eq CompCopLoa2 = identity(CompCopLoa1) \eq
identity(CompCopLoa2)
..
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%specification of the composition CompCopLoc between Classes: Copy and
LocalUse
%larch file: CompCopLoc.lp
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


set name CompCopLoc

declare sorts CompCopLoc, Copy, LocalUse, Nat, SetCopy, SetLocalUse
declare variables CompCopLoc1, CompCopLoc2 : CompCopLoc, aCopy1, aCopy2
: Copy, aLocalUse1, aLocalUse2 : LocalUse
declare variables SetCopy1, SetCopy2 : SetCopy, SetLocalUse1,
SetLocalUse2 : SetLocalUse, i : Nat

declare operators
%generator of an empty association
void :      Nat   -> CompCopLoc

%association identity
identity :  CompCopLoc  -> Nat
__\eq__  :  CompCopLoc, CompCopLoc  -> Bool

%operators to create and remove links
addLink : CompCopLoc, Copy, LocalUse -> CompCopLoc
removeLink : CompCopLoc, Copy, LocalUse -> CompCopLoc

%operator to test if two instances are linked
isLinked : CompCopLoc, Copy, LocalUse -> Bool

%operator to test if one instance is linked
isLeftLinked : CompCopLoc, LocalUse -> Bool
isRightLinked : CompCopLoc, Copy -> Bool

%operator to test if the association is Empty
isEmpty : CompCopLoc -> Bool

%operator to return the Set of instances linked to a given instance
allLeftLink : CompCopLoc, LocalUse -> SetCopy
allRightLink : CompCopLoc, Copy -> SetLocalUse

%operators for multiplicity values
leftMultiplicity : CompCopLoc, LocalUse -> Nat
rightMultiplicity : CompCopLoc, Copy  -> Nat

%special operator for Composition
isPartOf : CompCopLoc, Copy, LocalUse  -> Bool
..

% axioms
assert
sort CompCopLoc generated by void, addLink;

%axiom to state that tuples of instance values cannot be equal in an
association
```

```
(aCopy1 \eq aCopy2) /\ (aLocalUse1 \eq aLocalUse2) =>
addLink(addLink(CompCopLoc1, aCopy1, aLocalUse1), aCopy2, aLocalUse2) =
addLink(CompCopLoc1, aCopy1, aLocalUse1);

%axioms for the isEmpty operation
isEmpty(void(i));
~(isEmpty(addLink(CompCopLoc1, aCopy1, aLocalUse1)));

%state when two instances of object types are linked
~(isLinked(void(i),aCopy1, aLocalUse1));
isLinked(addLink(CompCopLoc1, aCopy1, aLocalUse1),aCopy2, aLocalUse2) =
((aCopy1 \eq aCopy2 /\ aLocalUse1 \eq aLocalUse2) \/
isLinked(CompCopLoc1, aCopy2, aLocalUse2));

%state the semantics for the operations allLeftLink and allRightLink
through addLink generator
(aLocalUse1 \eq aLocalUse2) => allLeftLink(addLink(CompCopLoc1, aCopy1,
aLocalUse2), aLocalUse1) = insert(aCopy1, allLeftLink(CompCopLoc1,
aLocalUse1));
(aCopy1 \eq aCopy2) => allRightLink(addLink(CompCopLoc1, aCopy2,
aLocalUse1), aCopy1) = insert(aLocalUse1, allRightLink(CompCopLoc1,
aCopy1));
~(aLocalUse1 \eq aLocalUse2) => allLeftLink(addLink(CompCopLoc1,
aCopy1, aLocalUse2), aLocalUse1) = allLeftLink(CompCopLoc1,
aLocalUse1);
~(aCopy1 \eq aCopy2) => allRightLink(addLink(CompCopLoc1, aCopy2,
aLocalUse1), aCopy1) = allRightLink(CompCopLoc1, aCopy1);

%state the semantics for allLeftLink and allRightLink through void
generator
allLeftLink(void(i), aLocalUse1) = {}: SetLocalUse;
allRightLink(void(i), aCopy1) = {}: SetCopy;

%state when one instance is linked through the association
~(isLeftLinked(void(i), aLocalUse1));
~(isRightLinked(void(i), aCopy1));
isLeftLinked(addLink(CompCopLoc1, aCopy1, aLocalUse1), aLocalUse2) =
((aLocalUse1 \eq aLocalUse2) \/ isLeftLinked(CompCopLoc1, aLocalUse2));
isRightLinked(addLink(CompCopLoc1, aCopy1, aLocalUse1), aCopy2) =
((aCopy1 \eq aCopy2) \/ isRightLinked(CompCopLoc1, aCopy2));

%axioms for left and rightMultiplicity operators
leftMultiplicity(void(i), aLocalUse1) = 0;
rightMultiplicity(void(i), aCopy1) = 0;
(aLocalUse1 \eq aLocalUse2) => leftMultiplicity(addLink(CompCopLoc1,
aCopy2, aLocalUse2),aLocalUse1) = leftMultiplicity( CompCopLoc1,
aLocalUse1) + 1;
~(aLocalUse1 \eq aLocalUse2) => leftMultiplicity(addLink(CompCopLoc1,
aCopy2, aLocalUse2),aLocalUse1) = leftMultiplicity( CompCopLoc1,
aLocalUse1);

(aCopy1 \eq aCopy2) => rightMultiplicity(addLink(CompCopLoc1, aCopy2,
aLocalUse2),aCopy1) = rightMultiplicity( CompCopLoc1, aCopy1) + 1;
~(aCopy1 \eq aCopy2) => rightMultiplicity(addLink(CompCopLoc1, aCopy2,
aLocalUse2),aCopy1) = rightMultiplicity( CompCopLoc1, aCopy1);
```

```
%axioms for multiplicity constraints: written only if multiplicity is
not free, i.e. different from 0 or More
%source multiplicity is 1 or more
%target multiplicity is Just One
~(size(allLeftLink(CompCopLoc1, aLocalUse)) < 1;
size(allRightLink(CompCopLoc1, aCopy1)) = 1;

%special axioms for Composition
isPartOf(addLink(CompCopLoc1, aCopy1, aLocalUse1), aCopy2, aLocalUse2)
=> (aCopy1 \eq aCopy2) /\ (aLocalUse1 \eq aLocalUse2);
~(isPartOf(void(i), aCopy1, aLocalUse1);

~(aLocalUse1 \eq aLocalUse2) => (isPartOf(CompCopLoc1, aCopy1,
aLocalUse1) /\(~(isPartOf(CompCopLoc1, aCopy1, aLocalUse2)))) \/
(isPartOf(CompCopLoc1, aCopy1, aLocalUse2) /\(~(isPartOf(CompCopLoc1,
aCopy1, aLocalUse1))));

%axioms for association identity
identity(void(i)) = i;
identity(addLink(CompCopLoc1, aCopy1, aLocalUse1)) =
identity(CompCopLoc1);
CompCopLoc1 \eq CompCopLoc2 = identity(CompCopLoc1) \eq
identity(CompCopLoc2)
..

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%specification of the identifier type for Library isntances
%larch file: IdLibrary.lp
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

set name IdLibrary

declare sorts IdLibrary
declare variables : idLibrary1, idLibrary2 : IdLibrary
declare operators
newIdLibrary :   -> IdLibrary
nextIdLibrary : IdLibrary  -> IdLibrary
__\eq__ : IdLibrary, IdLibrary  -> Bool
..

assert
sort IdLibrary generated by newIdLibrary, nextIdLibrary;
newIdLibrary \eq newIdLibrary;
~(newIdLibrary \eq nextIdLibrary(idLibrary1));
~(nextIdLibrary(idLibrary1) \eq newIdLibrary);
nextIdLibrary(idLibrary1) \eq nextIdLibrary(idLibrary2) = (idLibrary1
\eq idLibrary2);
..

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%
%specification of the type Set for User Class
%larch file: SetUser.lp
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%
```

```
set name SetUser

declare sorts User, SetUser, Nat
declare variables aUser1, aUser2 : User, setUser1, setUser2 : SetUser
declare operators
{} :  -> SetUser
{__} : User -> SetUser
insert : User, SetUser -> SetUser
__\U__ : SetUser, SetUser -> SetUser
__\in__ : User, SetUser -> Bool
__\I__ : SetUser, SetUser -> Bool
size : SetUser -> Nat
..

assert
sort SetUser generated by {}, insert;

{aUser1} = insert(aUser1, {});
~(aUser1 \in {});
aUser1 \in insert(aUser2, setUser1) <=> (aUser1 \eq aUser2 \/ aUser1
\in setUser1);
{} \I setUser1;
insert(aUser1, setUser1) \I setUser2 <=> (aUser1 \in setUser2 /\
setUser1 \I setUser2);
aUser1 \in (setUser1 \U setUser2) <=> (aUser1 \in setUser1 \/ aUser1
\in setUser2);

% axioms for size operator
size({}) = 0;
(aUser1 \in setUser1) => size(insert(aUser1, setUser1)) =
size(setUser1);
~(aUser1 \in setUser1) => size(insert(aUser1, setUser1)) =
1+size(setUser1);
..
```