OCL Evaluation Framework
Integrated Environment for
Syntactic and Semantic Evaluation of
OCL Constraints

Peter Levchenko

Final year project

Supervisor
Dr. Alessandra Russo

Second Marker
Dr. Francesca Toni

June 18, 2008
Acknowledgements

I thank Dr. Alessandra Russo for supervising my project. Her critical assessment of my work at every meeting has helped me to improve and broadened my awareness of potential alternatives to design decisions I have made. On a personal level, I really appreciate her friendliness, positive focus and persistent support.

I thank Dr. Francesca Toni for sparking the idea of semantic evaluation in my head and agreeing to be my secondary marker. I thank her for teaching me logic and her willingness to engage in argumentation with me during tutorial sessions, which has greatly stimulated my understanding of the subject.

I thank Dr. Krysia Broda for her critical evaluation of my goal-directed prover, which has encouraged me to extend and improve it significantly. I would also like to thank her for excellent teaching of Reasoning About Programs course, which has made me a more critical and logical person in general and improved my attention to semantics of things said, written and implied.
Abstract

OCL is ideal for defining constraints on UML models. None of state-of-the-art OCL software provides acceptable support for OCL: syntactic evaluation is mostly erroneous and incomplete for latest OCL 2.0 specification, while automated semantic evaluation, such as determining satisfiability of OCL constraints, is practically non-existent.

This report documents creation of OCL Evaluation Framework, which provides almost complete and practically usable syntactic evaluation for OCL 2.0 and can perform semantic evaluation of all OCL expressions, which involve non-collection types.

We also provide a simple user front-end to demonstrate functionality of OCL Evaluation Framework.
## Contents

1 Introduction .................................................. 3  
   1.1 Motivation ............................................. 4  
   1.2 Contributions ......................................... 5  
   1.3 Overview .............................................. 6  

2 Background .................................................. 7  
   2.1 Metamodels ............................................ 7  
      2.1.1 Unified Modeling Language (UML) ............... 8  
      2.1.2 Object Constraint Language (OCL) ............. 11  
      2.1.3 MOF, UML and OCL Metamodels ................. 19  
      2.1.4 XML Metadata Interchange (XMI) ............... 24  
   2.2 Syntactic Analysis .................................... 24  
   2.3 Semantic Evaluation in OCL ......................... 25  
      2.3.1 Existing approach: HOL-OCL .................... 26  
      2.3.2 Goal-Directed Theorem Prover for Classical Logic 28  
   2.4 Underlying Technology ............................... 31  
      2.4.1 Poseidon for UML ............................... 32  
      2.4.2 Java Metadata Interface (JMI) .................. 32  
      2.4.3 NetBeans Metadata Repository (MDR) ......... 32  
      2.4.4 SICStus Prolog .................................. 32  
      2.4.5 Wolfram Mathematica ............................ 33  
      2.4.6 SableCC Parser Generator ..................... 33  
   2.5 Evaluation of State-of-the-Art OCL Software ...... 34  
      2.5.1 Dresden OCL Toolkit ............................ 34  
      2.5.2 LCI OCL Environment (OCLE) ................... 36  
      2.5.3 Octopus ......................................... 38  
      2.5.4 Conclusions ..................................... 39  

3 Specification and Design: Syntactic Evaluation ...... 40  
   3.1 Limitations of OCL .................................. 40  
   3.2 Specification of the Subsystem ...................... 42  
   3.3 Overall View of the Design ......................... 43
Chapter 1

Introduction

Many software projects today require complex rules, written for specific business models, which need to be passed between designers and developers. This stimulates a need for a common standard, which enables developers to write precise specifications, constraints and queries over software models.

Object Constraint Language (OCL) satisfies this need as it enables developers to specify expressions and constraints on object-oriented models, which are defined in an industry standard such as Unified Modeling Language (UML is detailed in Section 2.1.1).

We will motivate the need for OCL using the following simplified example. Consider a situation that may arise in a specification of a flight management software system (Figure 1.1 shows the corresponding class diagram).

![UML Model for the Flight Control System](image)

**Figure 1.1: UML Model for the Flight Control System**

**Example 1.1:** System designer wishes to specify that at any one time the total number of flights in the airport (both, departing and arriving) should not exceed 60. This requirement needs to be passed to the implementers together with the UML model (given in Figure 1.1). In stand-alone UML this is not possible. One can use OCL, which is a subset of UML, (OCL is discussed in greater detail in Section 2.1.2) to specify this constraint and attach it to the UML model.

The constraint could be specified as an invariant in OCL (an expression which always holds true in the system):

```
context FlightCentre
   inv: self.flights.size() <= self.MAXFLIGHTS
```
It states that the number of association ends for directed association “controls”, which effectively holds all the flights, should never exceed 60. This requirement could be passed to the implementers by adding the OCL constraint to the UML model (UML explicitly supports this), as in Figure 1.2.

![UML Model for the Flight Control System, augmented with OCL constraints](image)

Figure 1.2: UML Model for the Flight Control System, augmented with OCL constraints

Although this constraint could be specified in some linguistic form, this can, in general, result in ambiguous and unclear specifications and inconsistent notation. OCL provides advantages of a precise, and formal language for specification of constraints with a well-defined syntax, which could be verified automatically by software.

### 1.1 Motivation

So far, we have motivated the need for OCL in modern software development. However, the current use of OCL is limited mostly to academics, rather than businesses. One of the main reasons for this is the absence of error-free, user-friendly and comprehensive OCL tools which provide sufficiently powerful environment for creating, editing and evaluating constraints and expressions.

Object Constraint Language by definition underpins a joint object-oriented framework, which would allow one to define UML class models, specify and check OCL constraints syntactically, and evaluate them against the corresponding UML model. In the context of Example 1.1 given above, there is currently no software, which would allow us to load the UML model from Figure 1.1, specify and check any well-formed OCL constraint on that model, and produce the resulting augmented UML model, which is given in Figure 1.2.

Although there are integrated software packages for UML, which also provide support of OCL, the scope of their OCL capabilities is extremely limited. **Problems with current OCL software** can be summarized as follows (State-of-the-art OCL tools are discussed in greater detail in Section 2.5):

- **Incomplete and erroneous support of the OCL syntax**, as specified by OCL standard (currently at version 2.0 [OCL20Spec])
- **Absence of correct and comprehensive type checking** (contextual analysis) of OCL elements for OCL expressions and constraints
• **Absence of object instance checking facilities**, which would enable one to load a model of object instances into the software and verify validity of specified OCL constraints with respect to the instance model.

• **Lack of extensive semantic evaluation of OCL expressions**. Usage of OCL can lead a user to specify contradictory requirements, which cannot all be satisfied simultaneously. There is almost complete absence of analysis of such situations available in state-of-the-art OCL tools.

• **Absence of an easy-to-use user-friendly interface** for OCL functionality in software. Most of the tools are not easy to grasp and do not present functionality in a convenient way. This makes OCL tools extremely unattractive to software developers.

The problems above motivate a need for a complete software package, which would provide a flexible integrated UML/OCL environment for editing and evaluating constraints. Availability of such software or integration of a stand-alone OCL package into existing UML software would encourage demand for use of OCL together with UML by software developers in a business context.

An attempt to solve all or at least most of these problems in order to bring the OCL standard closer to software engineering industry is the central idea which drives this project. We concentrate on developing a flexible software package, which would attempt to provide complete support of OCL 2.0 features, as well as extensive facilities for syntactic and semantic constraint evaluation.

### 1.2 Contributions

This report presents the development of software, called the OCL Evaluation Framework, which attempts to provide a comprehensive suite with capabilities for OCL. The contributions to the project are mostly motivated by the shortcomings of state-of-the-art OCL tools (these were outlined in the previous section) and include extensions for semantic evaluation, which is currently not implemented well in any OCL software.

The contributions of the project are:

- OCL Evaluation Framework has been developed which provides an environment for creating, viewing and editing OCL constraints.
- The developed software generates the complete integrated OCL/UML Metamodel from OCL constraints and associated UML class model. The common metamodel (Meta-models are described in Section 2.1) is produced from:
  - the UML class model, loaded by user
  - the OCL library, which contains built-in OCL types and operations (this library is described in Section 2.1.2).

  This capability is developed by extending the Metamodel-based OCL Compiler from Dresden OCL Toolkit (the latter is detailed in Section 2.5.1).

- We develop a complete system for syntactic evaluation of OCL constraints against UML class models, in accordance with the OCL 2.0 specification. It allows verification of both, grammatical and contextual correctness of user-defined OCL constraints. The system is developed by significantly extending OCLParser subsystem from Dresden OCL open source project.
• We detail and develop a system of semantic analysis in OCL Evaluation Framework, which determines **satisfiability and excessive-ness of OCL constraints and expressions**, specified on non-collection types. For each OCL constraint specified, OCL Evaluation Framework determines:
  – whether it contradicts other constraints
  – whether the given constraint is already implied by other constraints, i.e. the constraint is excessive

• We propose a flexible modular design for the system, which facilitates extensibility of both, syntactic and semantic evaluation subsystems in order to accommodate new and currently unimplemented features of OCL. This property relies on the modular design of the underlying Dresden OCLParser subsystem and on our implementation of the semantic evaluation component, which is almost entirely self-contained. OCL Evaluation Framework therefore allows one to substitute the semantic evaluation subsystem with minimum changes to the source code. The underlying system is also entirely decoupled from the Graphical User Interface, i.e. extending and entirely overhauling the GUI requires no changes to the underlying system.

### 1.3 Overview

We have thus outlined the main motivation and contributions of this project. This section gives a brief overview of what is to come in this report.

In the *Background* chapter, we introduce the notion of metamodels and describe the UML and OCL metamodels as well as their niche in the OMG’s model-driven architecture, known as Meta Object Facility ([Section 2.1](#)). Consequently, we define the notion of semantic evaluation in the context of OCL and introduce the theory behind goal-directed theorem provers in propositional logic ([Section 2.3](#)), which will underpin our implementation of semantic evaluation in OCL Evaluation Framework. This is followed by a description of the technology, which is we will require in the design and implementation of our system. The final section of the chapter presents and evaluates the state-of-the-art OCL software in order to identify critical shortcomings and thus highlight the significance of the functionality implemented in OCL Evaluation Framework.

*Specification and Design* chapter discusses the current limitations and problems associated with the OCL specification and, as a result, the extent to which the scope of our solution is constrained. It describes the functionality and design of all the subsystems (metamodel generation, syntactic and semantic evaluation), which constitute the OCL Evaluation Framework.

Consecutive chapters present critical challenges and details of the implementation of syntactic and semantic evaluation, followed by extensive testing in the form of a case study. We conclude the report with evaluation of the developed software and discussion of potential future work in the context of our software and the Object Constraint Language on the whole.
Chapter 2

Background

This chapter specifies the formal context of OCL in software engineering (Section 2.1), details capabilities that are currently provided for OCL through evaluation of current publicly available OCL software (Section 2.5) and describes technology, which underpins the development of OCL Evaluation Framework (Section 2.4). It also discusses existing approaches to semantic evaluation in the context of OCL (Section 2.3).

2.1 Metamodels

In software engineering, the notion of a metamodel refers to a collection of concepts (e.g. objects, classes, terms, definitions etc) within a common domain. Metamodels can be understood in the context of a chain of abstractions (as in Figure 2.1). In the context of our project, we define metamodel to be a schema which expresses additional semantics for a certain model (e.g. UML class diagram, or OCL language). An important requirement of a metamodel is that it contains data in a standardized format which can be exchanged and stored.

In modern software engineering, the use of models and metamodels is becoming increasingly encouraged. One of the most widespread approaches is Model-Driven Architecture (MDA) [MDASpec] proposed by Object Management Group (OMG). The central idea behind MDA, is that it separates business and application requirements from its underlying
technology. The specifications are defined using a platform-independent model, which can then be realized on technology platforms such as .NET, Java etc. The approach enables completely decoupled evolvement of business and technology models.

Platform-independent models are developed using Meta Object Facility (MOF) metamodel, proposed by OMG (this model is discussed in Section 2.1.3). In the context of software development, catering for metamodels, as opposed to models directly, offers clear-cut advantages:

- ability to process other models, which are instances of the implemented metamodel.
- software is easily extensible in response to changing specification of the underlying model; e.g. the new version of OCL might propose a significantly altered structure of OCL syntax. If a metamodel for OCL is used by the software, the only alterations required would involve manipulation of attributes of the metamodel, as opposed to a complete overhaul of existing implementation.

In the following sections, we describe the models for UML and OCL and produce the overall view of these models as part of the MOF metamodel.

2.1.1 Unified Modeling Language (UML)

UML is a formal language for modeling objects [UML2Spec], developed by OMG. It includes a framework of graphical notation, which can be used to specify abstract models of a software system. In the context of our project, we are only interested in UML class diagrams and object instance models.

Elements of a UML Class Object Model

Figure 2.2 illustrates a typical UML Class Diagram, which models the structure of classes in a software system. In this example, package “carworld” models a real world situation, e.g. Car World. It contains a nested package (“nested”), which serves the purpose of illustrating that one can define nested packages to model sub-situations within the overall situation being modeled. The global identifier of “nested” in this case is “example::carworld::nested” (where “example” identifies our UML class model).

The class diagram can contain several items, which are formally defined as model elements:

- Classifiers. These typically correspond to classes in the source code of the implementation of software. They represent a common entity, which will be used to produce object instances in the execution of software. In Figure 2.2 “Car” is an example of a classifier, which models the notion of a car in our Car World. A classifier can, in turn, include the following model elements:

  - Attributes. These correspond to properties of a class in the source code and represent features of the entity, modeled by the parent Classifier, e.g. “age” is an attribute of classifier “Person”, which models an age of a person in our Car World (represented by package “carworld”). UML is a typed language and each attribute has an associated type, e.g. type “int” (which is also a classifier, and the converse is true: UML classifiers are types) of attribute “age” models the notion that age of a person is an integer number.
Figure 2.2: Car Owner Example, extended with an association class
- **Operations**, which can be associated with the methods of a class in the source code and represent *functions* of the parent real world entity. For instance, “getTopSpeed()” is a function of classifier “Car”, which could model a driver looking at the speedometer of a car. Similar to attributes, operations also have types (return types), which define the formal UML type of the output from an operation. Type “void” indicates does not return anything to the caller.

An operation can also accept **Parameters**. E.g. “driveTo” operation takes hypothetical x and y parameters, which model position of some location in our Car World. Each parameter is associated with a **type** (x and y are of type “double”, which corresponds to a real number) in the same way as attributes are (which we discussed above).

- **Associations** model *relationships* between entities (classifiers in UML) in the real world situation. In source code of the software, this corresponds to classes containing references to other classes, which can, for instance, be used to call methods on other classes. E.g. “Production” is an association, which represents the notion that a car is built by a manufacturer (represented by classifier “Producer” in Figure 2.2). Properties of associations are modeled by other elements:

  - **Association Ends** denote participants in an association. E.g. association “Production” has two association ends with their names: “producedCars”, associated with classifier Car (i.e. this association end is of type carworld::Car), and “producedBy”, associated with “Producer”. In source code, classifier “Producer” maintains a reference to a collection of cars (in “producedCars”), while “Car” holds a reference to its manufacturer (in “producedBy”). Notice that one is a collection of instances, whilst the other is a single instance. We infer this information from

  - **Multiplicities**. “*” next to “producedCars” indicates that classifier “Producer” can hold a collection of “Car” of any size (0..n for some integer n). “1..*” would indicate that the minimum size of this collection is 1. “k” would indicate that in all possible instances of our model of Car World, any producer has built exactly k produced cars, for some integer k.

  - **Association Classes** can optionally be assigned to associations. It has the same underlying structure as a Classifier, and models properties (features via attributes and functions via operations) of the corresponding association. E.g. the only other association in Figure 2.2 has a corresponding association class “Purchase”, which models the notion that a person becomes the buyer of a car (association) through a purchase, which has a price (denoted by attribute “purchasePrice”) and the base price of the car model (given by “price”) together with a calculator of the actual price from the base price (e.g. based on customized parts and extras ordered by the buyer), which is modeled by operation “computePrice()“.

- **Visibilities** can be associated with classifiers, attributes, operations and association ends. They model *scope of access*, i.e. who can gain access to what. **Public** visibility indicates that the associated element can be accessed from any other model element, e.g. in our model, attribute “age” can be accessed from anywhere. This models the fact that, in particular, any producer can find out a buyer’s age in our Car World.

- **Constraints** represent additional semantic information, which can be attached to model elements in UML. A constraint models an assertion, i.e. a restrictive expression of some form, which must be satisfied by correct designs of the system, in accordance with the class model. Constraints are specified within a certain context, which
ultimately reduces to some UML element, referred to as constrained element. Stand-alone UML allows specification of constraints in any textual notation. In this project, however, we only consider OCL as a formal language of definition for constraints in UML.

For the sake of simplicity and more concise scope, we will not refer to other elements of UML in much detail (other visibilities, static attributes etc). For the full description of UML elements and notation see [UML2Spec].

UML Class Models vs Object Instance Models

An Object Instance model is an instance of a UML Class model. It can be used to represent one execution of the software system, which is represented by the corresponding UML Class model. Key differences from the UML class model are:

- initialized entities are modeled by objects, which are instances of classes from the UML class model
- attributes, association ends and multiplicities have associated initialized values, e.g. attribute “age” from our Car World example would have a value, e.g. age = 16.

UML class models are a standardized template of UML in that they have a formal definition, the UML Metamodel (described in Section 2.1.3). They are also portable via the XMI standard which enables one to use UML class models defined in one software package with a different package, although there are portability issues (these are detailed in Section 2.1.4 along with the XMI standard).

Although a UML object instance model could be defined using the UML template for a class model (e.g. using Poseidon for UML, which is detailed in Section 2.4.1) it does not have a widely accepted commonly agreed metadata representation such as XMI (although it can be described by the UML metamodel, this is shown in Section 2.1.3). This makes it impractical to transfer UML object instance models between software packages. Hence, the most obvious potential use of OCL, which is verifying constraints against object instance models, is impractical to implement and hence not supported by any state-of-the-art OCL software (discussed in Section 2.5). Although UML object instance models could be defined in MOF (which is described in Section 2.1.3) and XMI could in principle be used to exchange such models, the absence of commonly agreed definition of object instance models in either XMI or MOF constitutes the main reason why support of object instance models in OCL software is absent.

2.1.2 Object Constraint Language (OCL)

In this section we provide a brief introduction to the Object Constraint Language and its constructs.

OCL is a logic-like declarative language that provides two kinds of descriptions:

- constraints, which are restrictive statements with respect to some underlying model that should evaluate to true. E.g. OCL can be used to state “the total number of objects in the model cannot exceed 10”

- expressions, which are defined in the context of some constrained element of the underlying model (e.g. UML classifier) and evaluate to some value, which by definition is of the same type as the associated constrained element. E.g. OCL can be used to state “attribute “age” of all objects of type “Person” is greater than 6”
OCL was developed at IBM, but is now part of OMG’s standards, which enables it to be used with any metamodel, specified in MOF (Meta-Object Facility is described in Section 2.1.3).

The OCL 1.4 Specification defined OCL as a pure constraint language, but it has been significantly extended in 2.0 Specification (see [OCL20Spec]) to include query operations and expressions on objects. As a result, it can be used to express constraints on any model and any metamodel, defined in MOF, as well as on the MOF metamodel itself to define assertions, which cannot otherwise be specified by diagrammatic notation.

In the context of this project, we concentrate on the use of OCL in supplementing UML by providing specifications of requirements, which cannot otherwise be specified precisely, non- ambiguously and coherently within UML class models.

In the following sections we briefly detail the formal OCL types and operations provided by the OCL Library, and describe and exemplify the types of expressions OCL can be used to specify.

### Built-in OCL Types and Operations

Basic built-in OCL Types are presented in Table 2.1. Note that “...” denotes all the operations OCL provides for objects of any type (which are therefore attached to all the individual types listed in the table). The table specifies only the operations that have been implemented in our OCL Evaluation Framework.

Most of the operations are intuitive, such as all the logical connectives, and numerical relations = (equals), < (not equals), ..., abs (absolute value), div (quotient of division by a number), mod (remainder of division by a number), round (rounded value of a number), min (minimum of two numbers) and max. size() returns the size of a string, concat() joins two strings, toInteger() and toReal() would convert ’3’ into a number 3, toUpper() gives the string in uppercase, and toLower() - in lowercase.

We will be using let expressions to instantiate variables, that we use to define our statements in OCL. Specifications will be annotated with comments. The general form is as follows:

```ocl
let <var_name> : <var_type> = <var_value> in <expression>  -- <comment>
```

Let us exemplify some of the operations with the following OCL expressions:

```ocl
let a: Boolean = true in a and not a implies false  
    -- instantiates to true ^ -true -> false
```

<table>
<thead>
<tr>
<th>Type</th>
<th>Example Values</th>
<th>Predefined Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>true, false</td>
<td>and, or, xor, not,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>implies, =, &lt;&gt;, ...</td>
</tr>
<tr>
<td>Integer</td>
<td>1, -1, 2, 0</td>
<td>*, +, -, /, abs, div, mod,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>min, max, &gt;, &lt;=, &lt;, &gt;, &gt;=, =&lt;, ...</td>
</tr>
<tr>
<td>Real</td>
<td>1.5, -1, -2.4, 0</td>
<td>*, +, -, /, abs, div, mod, floor, round,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>min, max, &gt;, &lt;=, &lt;, &gt;, &gt;=, =&lt;, ...</td>
</tr>
<tr>
<td>String</td>
<td>'peter', 'ocl is great'</td>
<td>=, &lt;&gt;, size, concat, substring,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>toInteger, toReal, toUpper, toLower, ...</td>
</tr>
</tbody>
</table>

Table 2.1: Basic OCL Types
let b: Integer = 3 in -b.abs() / 3.max(2) = 1
-- instantiates to |-3| / max3,2 = 1
-- operations are always called in postfix notation

let c: Real = 3.4 in c.round() <> c.floor() + 1
-- instantiates to 3 <> c.floor() + 1
-- note how the ceiling of c can be
-- defined as c.floor() + 1
-- for this reason, we do not specifically
-- define operation ceil()

let d: String = 'ocl' in d.concat(' ').size() > d.size()
-- exhibits the fact that a space is a character

OCL also contains Collection Types:

- **Set.** e.g. Set {1, 2, 3, 66} is a mathematical set, i.e. it does not contain duplicate elements. It also does not have an order defined on the elements, hence Set {1, 3, 2} = Set {1, 2, 3}
- **Bag** is similar to a Set, except it may contain duplicate elements: Bag { 'ape', 'b', 'ape' }. The order on a Bag is not defined.
- **Sequence.** e.g. Sequence { 1, 2, 45, 2 } is like a Bag in that it can contain duplicates, but it does have a defined order, i.e. Sequence {1, 3, 2} <> Sequence {1, 2, 3}.

All the collection types above conform to the abstract type Collection. Type conformance is discussed further in the sub-section Extensions below.

Finally, OCL provides conventional object-oriented types such as

- **OclAny**, which is the supertype of any type used in OCL (i.e. all OCL types conform to OclAny)
- **OclUndefined**, which represents the type of a model element whose value has not been initialized
- **OclVoid**, which can be used e.g. to indicate that an operation does not have a return type.

We will exemplify the use of these types and other OCL constructs in due course.

**Context**

Expressions specified in the previous section are syntactically correct, but not contextually correct. This is because all OCL expressions must be specified within a certain context. Recall that OCL expressions are not stand-alone in that they must be defined in the context of some metamodel, which is the UML metamodel in our discussion (for a detailed description of the UML metamodel, see Section 2.1.3).

Consider again the Car Owner example given in Figure 2.2. It represents an instance of the UML metamodel. Hence, OCL expressions we have specified above could be associated with this instance:
package carworld -- specifies the name of the UML Package
context Person -- specifies the name of the context
    -- here, a UML Classifier

let a:Boolean = true in
age > 0 = a
-- this expresses an invariant (introduced later)

endpackage carworld -- package context declarations must
    -- end with endpackage

In this example, “context Person” specifies that the expression that follows is associated with an instance of type “Person”, not the class “Person” itself! This underpins the name of OCL as an object-based language. Class “Person” is referred to as a constrained element of the specified context.

Contexts in OCL can also be prefixed with a name and can always be referred to using the keyword “self”:

context pers:Person -- now a typical instance of Person can be
    -- referred to by 'pers'
pers.age > 0 and self.age > 0 -- the two conjuncts are equivalent

The expression states that the value of attribute age of an object of type “Person” is greater than 0. In the first conjunct, the object is referred to by the chosen context name “c”. Keyword “self” can be used to refer to the object irrespective of whether the context is prefixed with a name or not. In OCL, association ends, attributes, operations and association classes can act as constrained elements as well as Classifiers.

However, the choice of context largely depends on the type of the constraint being specified. The OCL expressions, which we have exemplified so far, are still specified incompletely. Every OCL expression has to be declared within a certain expression type. These are detailed in the following sub-sections.

Invariants

Invariants are constraints which specify that the associated expression is always true, i.e. true in all possible states of the system. In relation to the UML metamodel, an OCL invariant is a UML Constraint stereotyped as an “invariant”. An invariant must be declared in the context of a UML Classifier, e.g. “Person” which we will refer to as type. The OCL expression is then called an invariant of that type and must hold for all instances of that type at any time, e.g.:

context Car
    inv validModelName : self.model <> ''
    -- specifies the model name of ANY car must be non-empty

We have given the OCL constraint a name, “validModelName” (which is optional), the OCL expression follows after “:”. This would be annotated in the UML metamodel (exemplified in Figure 2.2) as an invariant of type Car, which specifies that ALL cars must have a well-defined model name in our Car World. Note that by definition OCL expression associated with an OCL invariant must evaluate to a Boolean.
Pre- and Post Conditions and Operation Body

Pre- and post- conditions correspond to “precondition” and “postcondition” stereotypes in the UML metamodel. They **must be declared in the context of a UML Operation.**

An OCL precondition is specified via an OCL expression which should must be satisfied (i.e. must hold true) before an Operation is entered. Hence if it is not specified, calling the operation is assumed to have no effect on the system. An OCL postcondition specifies a constraint which should be satisfied when the operation call has completed. A “result” variable can be used in the postcondition to refer to the returned output of the operation (if its return type is not OclVoid).

Several pre- and post-conditions can be defined within the same context as follows (the associated UML metamodel is instantiated in Figure 2.2 and is part of our Car World running example):

```uml
class Car {
  operation driveTo(d1 : Real, d2 : Real) : Boolean
    pre paramsOk : d1 > 0.0 and d2 > 0.0 and self.owner->notEmpty()

  operation getTopSpeed() : Real
    body query : self.topSpeed
    post : result = self.topSpeed and self.topSpeed = self.topSpeed@pre
}
```

This example exhibits several important points:

- pre-, post- and body- constraints can have names the same way as invariants
- operation `driveTo()` takes two arguments and returns a boolean, while operation `getTopSpeed()` takes no arguments and returns a real; in our Car World example it models a getter function for attribute `topSpeed`.
- the pre-condition asserts that the specified coordinates d1 and d2 are non-negative. It also states that the car must be owned (have at least one owner) before it can be driven in our Car World. The latter is expressed via “self.owner” which explores the Ownership association from an instance of “Car” in the UML metamodel through the association end named “owner”. Since the multiplicity of owner is “*”, a Set of type Person is returned (i.e. all associated persons should be accessible in UML). A call to an operation defined on Collections is always preceded with “→” and in this case “notEmpty()” operation, which is defined on Set, is called. It evaluates to true if the underlying set is not empty and false otherwise. Hence, the meaning it specifies.
- Suppose instead that the multiplicity of association end “owner” was exactly 1, then an object of type “Person” as opposed to a Set of “Person” would be navigated. This is an interesting example, since the call “→notEmpty()” is now syntactically invalid, because the underlying type is non-Collection. OCL, however, still allows this syntax assuming that the underlying instance of “Person” is implicitly converted to a singleton Set of Person, which contains the instance, via an implicit call to a pre-defined OCL operation `asSet()`. Hence, the corresponding explicit syntax is in reality:

```ocl
define self.owner->asSet() ->notEmpty()
```
- the post-condition on `getTopSpeed()` specifies that the result of the operation is the value of the attribute `topSpeed` of the instance of type Car on which operation is called. It also specifies that the value of attribute `topSpeed` is not changed by the operation, i.e. it is a query operation. This is specified by using the suffix `@pre` which references the value of the attribute before the operation is executed.
The OCL expression prefixed with a “body:” tag denotes an OCL operation body definition. This constraint can be used to specify the result of the constrained query operation (one that does not change the internal state of the system). In fact, the body definition specified above is equivalent to the first conjunct of the postcondition. It specifies that the query operation `getTopSpeed()` returns the value of the attribute `topSpeed`.

**Initial and Derived Value Expressions**

An OCL expression can specify so called initial and derived values of UML attributes and association ends.

An initial value expression specifies the value of the constrained attribute/association end at initialization of instances in the system. A derived value expression specifies the value, which the constrained attribute/association end should always evaluate to. To exemplify:

```ocl
class Person{
  attribute Integer age;
}
context Person::age : Integer
init: 6  -- possible initial age
derive: if self.ownedCar->isEmpty() then 6  -- any age under 18
        else 18  -- at least 18,
        --since he owns at least one car
        endif
```

This example is defined on an attribute age (of type Integer), present within instances of type “Person” in the associated UML metamodel of our Car World example (Figure 2.2). It specifies that by default our system assigns all persons the minimum age of 6 in our Car World. This is asserted by the initial value expression. If the person owns at least one car, then the minimum age is updated to 18, since he must be at least 18 by the legislation in our Car World. This is asserted by the derived value expression by means of the if-then-else construct: we navigate to all the cars this instance of “Person” owns via `self.ownedCar` (which obtains Set of “Car”). A collection-type call “→isEmpty()” evaluates to true if there are no associated instances of “Car”, i.e. the value of attribute age is unchanged. If the call evaluates to false, then at least one “Car” must be owned by this instance of “Person”, i.e. the age should be updated to at least 18. Note that if-then-else expression may not have an else part but must always be terminated with `endif`.

To fully understand the semantics of the example above, consider the following initial state of the system:

- all instances of Person in the system have their attribute “age” instantiated to 6.
- some persons (at least one is sufficient) owns a car (i.e. its Set of “Car” given by navigating to association end “ownedCar” is not empty)

In this case, the derived value expression for “age” would evaluate to 18, while all the initial values of “age” in the system are set to 6. This would generate an unsatisfiable set of constraints, since the derived value expression for an attribute has to be satisfied by actual values in the system in all possible states.

**Attribute and Operation Definitions**

OCL allows defining custom attributes and query operations which are not part of the underlying UML metamodel. They become UML Constraints with stereotype “definition”, which is used to define helper methods and variables. Suppose we want to extend description
of cars in our Car World example with tuning information, i.e. whether a car has been tuned. This can be done purely in OCL as follows:

```ocl
class Car
  attribute tuned : Boolean = false -- not tuned by default

  operation isTuned() : Boolean = self.tuned
    -- query whether a car has been tuned
    -- (e.g. by AMG for Mercedes)
end class Car
```

We have given the OCL definition expressions names “attrExtra” and “operExtra”. Formal declarations of types, and any function parameters are required. We set all cars to be non-tuned by default in our Car World and specify a query getter operation which indicates whether a given instance of “Car” is tuned.

### Extensions

The description of OCL we have provided up until this point is incomplete. In this subsection we will define several other important elements of the OCL syntax that will be of use in further discussion in this report. However, for the complete specification please refer to [OCL20Spec](#).

OCL allows navigation of association classes in exactly the same way as UML Classifiers. The only exception is when we access association class via a qualified dot notation - we use the association class name in lowercase. In our example in Figure 2.2, we could navigate “Purchase” as follows:

```ocl
class Car
  invariant: self.purchasedCar[buyer] -> asSequence()
    -> first().age < 60
end class Car
```

Association class can be accessed via dot notation from any of the two associated Classifiers (in this case, “Car” and “Person”). In the example, the association is bi-directional which results in two instances of “Purchase” - one which models a relationship from `buyer` to `purchasedCar` and one which models the converse. How do we determine which instance is being called? In this case, qualification “buyer” is not required since in the context of Car, one can only navigate to association end of type “Person”. Hence, we could have simply used “`self.purchasedCar`” – a call to `asSequence()` converts a Set of “Person” (owners associated with our instance of “Car”) into a Sequence. It is critical to note that the order is defined as unpredictable when the operation is applied on a Set, hence the consequent call to `first()` can potentially result in any one instance of “Person” being returned. Thus the constraint specifies that potentially any owner of a car is less than 60 years old in our Car World.

The qualified notation becomes necessary when the corresponding association of the Association Class is recursive, i.e. it associates instances of the same type (e.g. “Car” to “Car”). Suppose we wanted to define a family relationship between persons in our Car World example. This would require associations between instances of the same type, “Person”. Then a call to a hypothetic association class “Family” via `self.family` within `context Person` would be ambiguous: are we trying to navigate from parent person to child person or vice
versa? If association end names were “parent” and “child”, we would write `self.family[parent]` to navigate from `child “Person”` to `parent “Person”`.

Within the framework we currently specified, it is hard to find a genuine use for Collection types outside navigation of association ends and association classes. However, OCL library also provides the `allInstances()` operation which allows to execute queries on collections of objects similar to database query languages:

```plaintext
context Person
inv: Person.allInstances() -> select (p:Person | p.age > 70)
    -> isEmpty()

inv: Person.allInstances() -> select (age <= 70)-> size() = Person.allInstances()->size()
```

`allInstances()` returns a Set of all objects in the system, which are instances of the underlying type (“Person” in the example above). The two invariants specify equivalent requirements: there are no persons in our Car World, who are older than 70. The first invariant achieves the specification by retrieving all instances of “Person”, and selecting a subset, for which attribute age is greater than 70. This set is then explicitly constrained to be empty. The second invariant retrieves a subset of all “Person” instances and asserts that the size of this subset is the same as the size of a set of all “Person” instances, i.e. in set notation (P is returned by `allInstances()`):

\[
\|P_{\text{age} \leq 70}\| = \|P\| \Rightarrow \|P_{\text{age} \leq 70}\| = \|P_{\text{age} \leq 70} \cup P_{\text{age} > 70}\| \Rightarrow P_{\text{age} \leq 70} \cap P_{\text{age} > 70} = 0.
\]

OCL provides an object-oriented framework for manipulating OCL types, which is referred to as Type Conformance. We have already mentioned the fact that all concrete Collection types, such as Set, Bag and Sequence conform to the same abstract supertype Collection. In OO programming languages, the notion of type conformance corresponds to casting. Evidently, e.g. Integer conforms to Real, and Set can be upcasted to Collection, and then downcasted back to Set. Furthermore, the following general type conformance rules hold:

- **TypeA conforms to TypeB** iff EITHER TypeA and TypeB are identical OR TypeA is a subtype of TypeB
- **collectionType1(Type1) conforms to collectionType2(Type2)** iff Type1 conforms to Type2 and EITHER collectionType2 is identical to collectionType1 OR collectionType2 is of type Collection

Examples:

- Set(Person) conforms to Collection(Person)
- Set(Car) conforms to Set(Transport) iff Car is a subtype of Transport
- Set(Car) conforms to Collection(Transport) iff Car is a subtype of Transport
- BUT Set(Car) conform to neither Bag(Car) nor Sequence(Car) because they are all on the same level of the collection type hierarchy and all subtype abstract supertype Collection.

Finally, all types of the integrated OCL/UML metamodel conform to type OclAny. The similar type in Java would be Object, since all types subclass Object. Within OCL type conformance can be evaluated using the following operations (which are defined on ALL objects):
• `oclIsTypeOf(OclType)`  e.g. context Person inv: self.oclIsTypeOf(Person) evaluates to true

• `oclIsKindOf(OclType)`  e.g. context Mercedes inv: self.oclIsKindOf(Car) is true (Mercedes subtypes Car)

• `oclAsType(OclType)` can be used for upcasting and downcasting. Consider an example for our Car World (Figure 2.3):

```
context Purchase
    inv: self.buyer.oclAsType(Collection)->notEmpty()
```

We navigate from association class Purchase towards association participant buyer of type “Person”. We upcast Set(Person) returned to a Collection(OclAny). Note that “notEmpty()” operation is called via definition for Collections, not for Sets. e.g. after upcasting to Collection, we would not be able to call, say operation union(), which is defined on Sets, but undefined for Collections.

OCL Type Conformance framework provides much greater flexibility than described here (for a full description see [OCL20Spec]).

2.1.3 MOF, UML and OCL Metamodels

Up until this point, we have described UML and OCL models using concrete notation, i.e. the notation one would actually use to develop instances of these models. E.g. our Car World in Figure 2.2 exemplifies an instance of UML. There is also more general notation we could employ to describe our Car World which is irrespective of the names we assign to our classes, which classes contain which attributes etc. This general notation is defined by the UML metamodel. In the case of OCL, we could use OCL metamodel, i.e. common notation, to describe all of the examples above as opposed to using different syntactic terms for each example (which is what we have done with OCL syntax). UML and OCL metamodels are in turn defined by the same “super”-metamodel, called the Meta-Object Facility (MOF), which is part of OMG’s standard. This section briefly presents the MOF metamodel and exemplifies UML and OCL metamodels.

MOF Metamodel

MOF is a model-driven engineering standard, which was originally designed by OMG as they were looking to develop notation, in which UML could be defined. As a result, MOF provides a four-layered architecture, which allows to specify intermediate metamodels, which use metadata to model a real-world problem. Figure 2.3 exhibits the four-layer architecture, which is better described using a bottom-up approach. At the lowest layer M0, we have real-world problems that we are trying to solve by designing software. These situations can be modeled using M1 layer models, e.g. we used a UML class diagram to model our Car World and defined OCL expressions to model certain facts about our Car World which cannot be expressed purely diagrammatically (however, our OCL model has references the underlying UML class model, as shown in Figure 2.3). In general, contents of layer Mn is modeled by contents of layer M(n+1). UML models, such as class diagrams, are defined by the common UML metamodel, which we detail in the next subsection; while OCL expressions (models), are defined by the OCL metamodel, which has dependencies on the UML metamodel elements (e.g. model elements, instances of UML.ModelElement, such as
The UML and OCL metamodels are in turn defined by the MOF metamodel, which in this context becomes a meta-metamodel (i.e. a model “about” metamodels). The structure and notation of the MOF metamodel itself is beyond the topic of this project (see [MOF2Spec] for a complete description).

Latest official version of MOF is 1.4 which corresponds to UML metamodel definitions 1.4 and 1.5. These are the latest UML versions currently supported by state-of-the-art OCL software. Although UML 2.0 has recently been released which uses an updated version of the MOF metamodel, the MOF 2.0 specification has not been completed yet and is only going through final pre-release stages.

**UML Metamodel**

The UML metamodel occupies M2 layer of the MOF architecture. It is defined by the MOF metamodel, which resides in layer 3 (this is shown in Figure 2.3). In this section, we start at the bottom M0 layer of the MOF architecture exemplify the modeling process through M1 and M2 layers, the latter exhibiting the UML metamodel itself.

Suppose we are modeling a simplified part of biological classification. **Human Example:** *Human is a species, which has two legs.* We have just provided the M0 Layer specification, i.e. a real-world situation. The corresponding M1 Layer model would be a UML class diagram, specified in Figure 2.4. “Human is a species” corresponds to a *generalization* in the UML class model, and the notion that it has legs corresponds to a strong association with the semantics of “contains/is composed of”, which is modeled by *composition*. The *multiplicities* denote the fact that each human has two legs. Species is modeled as an *abstract* classifier,
since we cannot allow instances of species, just like no species is classified as just “species” in the real world. Italicized terms are UML elements which are formally defined as part of the UML metamodel. The M2 layer UML metamodel defines semantics associated with model elements used in the M1 layer UML model. In fact, the model exhibited in Figure 2.4 can be represented by an instance of the UML metamodel, which is shown in Figure 2.5. Each

![Diagram](image)

Figure 2.5: Human Example as an instance of the M2 layer UML metamodel

model element of Figure 2.4, the UML class model, is shown as an object in Figure 2.5. The objects contain attributes that describe the semantics of the UML M1 layer model, e.g. “aggregation” attribute of AssociationEnd object denotes the type of association (which is “composite” for composition between “Human” and “Leg”) and “multiplicity” specifies the multiplicity of the association end in the association, which it participates in.

Figure 2.5 shows an instance of the M2 layer UML metamodel in that it is an object diagram - all the entities are instantiated classes, attributes and associations. The corresponding structure of the actual UML metamodel is shown as a class diagram in Figure 2.6. ModelElement, Namespace etc are classes of the UML metamodel, also known as meta-

![Diagram](image)

Figure 2.6: Part of the M2 layer UML metamodel used in Human Example (Source: [UMLMetaModel])
classes. Any UML model at M1 layer can be defined using instances of these metaclasses, and dependencies between them.

We have thus exemplified only a part of the UML metamodel (for a complete description see UML Semantics section of the UML specification [UML2Spec]).

OCL Metamodel

In this sub-section we briefly discuss the OCL Metamodel, the M2 layer model, by instantiating which one can specify any set of well-formed OCL expressions. Similar to our discussion of the UML metamodel above, we show how an OCL expression can be represented by the OCL metamodel. Suppose we define the following constraint on the UML class model in Figure 2.4 showing the Human Example (assume the association end name at “Leg” is +legs):

```ocl
class Human
  inv: self.legs->notEmpty()
  -- asserts that a Human must have at least one leg
```

This OCL expression would be assigned to class Human as a Constraint with stereotype “invariant”. The OCL expression itself can be represented using an instance of the OCL metamodel (at the M2 layer of the MOF architecture) as shown in Figure 2.7.

![Figure 2.7: “self.legs->notEmpty()” as an instance of the M2 layer OCL metamodel](image)

OperationCallExp, NavigationCallExp and VariableExp are all subclasses of OclExpression, which is the superclass of all OCL expressions. The OCL metamodel has dependencies to the “Core” package which extends classes of the associated UML metamodel, also at the M2 layer. In the example, the OCL metamodel references AssociationEnd, Operation and Classifier instances from the UML metamodel. AssociationEnd instances would ultimately
reference corresponding instances of the UML metamodel (an example of such instance is shown in Figure 2.7 for the Human Example), whereas Classifier and Operation can potentially refer to built-in types and operations of the OCL Library as well as the underlying UML metamodel.

The overall structure of the UML metamodel (as a class diagram), is shown, similarly to the already illustrated UML metamodel, in Figure 2.8.

Figure 2.8: The overall view on the M2 layer OCL metamodel (Source: [OCLMetaModel])

OclExpression is attached to a ModelElement in associated UML metamodel. It is subclassed by concrete types such as LiteralExp (e.g. "5", "peter"). IfExp (if-then-else constructs), VariableExp (variable declarations, e.g. a:Real = 3.5 in a let-in expression, self is a variable declaration, which references instances of a certain type - specified in "Classifier"). LoopExp, which allows collection-type calls such as select(), that may contain nested variable declarations (e.g. \( \rightarrow \text{select}(p:\text{Person} - p.\text{age}>50) \)) defines a nested variable “p”).ModelPropertyCallExp is an abstract subtype which is extended by all possible calls that can be made in OCL: referring attributes, operations, association ends, associations and association classes, which are typically part of the underlying UML metamodel. Note that there can also be calls to pre-defined operations in OCL, which means that package “Core” must reference the integrated OCL/UML metamodel as opposed to either OCL or UML.
2.1.4 XML Metadata Interchange (XMI)

So far, we have described the UML and OCL metamodels, and defined their niche in the MOF metadata modeling architecture. In this section we outline the standard, which is used for transferring metadata and which can therefore be used to transfer MOF metamodels and any metamodels and models, defined using MOF.

XMI is a standard for exchanging metadata of abstract and concrete models via XML. By OMG’s specification abstract models represent semantic information (e.g. instances of metamodels such as UML and OCL) while concrete models represent visual diagrams. XMI’s purpose is to support interchange of abstract models, which are defined in notation commonly referred to as metadata. Concrete models can be exchanged using the Diagram Interchange standard, defined within the XMI framework, but it is currently hardly ever used.

At this time, there are significant incompatibilities between different implementations of XMI, and even between definitions of abstract models using metadata. Hence, exchanging abstract models using XMI between different UML tools frequently results in errors and is, in general, impractical. In the context of OCL, one is interested in exchanging UML object instance models in addition to standard class models. However, as already discussed above, there is currently no agreed upon abstract model, and even if there existed one, XMI incompatibilities make it virtually impossible to transfer such models and hence prevent their widespread use.

We have thus described the notion of metamodels and presented UML and OCL in the context of the MOF model-driven infrastructure. The UML and OCL metamodels will become extremely relevant in our discussion of the design and implementation of the OCL Evaluation Framework. We have also outlined the XMI standard for exchange of metadata. We will refer to the standards presented in this subsection again, when discussing the relevant underlying technology such as NetBeans MDR Repository and Java Metadata Interface (see Section 2.4).

2.2 Syntactic Analysis

In this section we briefly outline the main aspects of syntactic analysis, knowledge of which will be necessary to follow the design of syntactic evaluation subsystem of OCL Evaluation Framework.

Syntactic analysis is a procedure, which ultimately transforms a linear representation of text, which conforms to some formal language, e.g. OCL constraints entered by the user conform to OCL syntax, into a structured representation of the text, that also maintains some semantics about the text. This semantics might be useful for operations to be performed on the text.

The series of steps which defines this procedure can be described as three main phases (adapted from [DresdenParser]).

1. Lexical Analysis. Characters of the input stream, which represents the input text, are matched against a set of rules and aggregated into tokens. This is done by the lexer component of the software. The rules for tokens are typically specified as regular expressions (regexes) to the lexer, and it produces a token stream as the final output.

2. Syntactic Parsing involves matching the token stream against a set of rules, referred to as grammar and is typically performed by a parser. The grammar consists of
productions. Each production can involve tokens, recursion and other productions, and can be visually represented as a tree. The rules are specified in the form of context-free grammar (CFG), which describes how expressions are built up from their smaller blocks, but cannot express relations such as reference of features to one another and agreement between features, which are not directly related by productions. The tree, which is built up to represent productions is referred to as the Concrete Syntax Tree. The latter is transformed further to Abstract Syntax Tree (AST) which is a form which carries over the semantics of the underlying text, but drops the text’s formal structure, which was only required by production rules of the grammar.

3. Contextual Analysis. Limitations of context-free grammar, mentioned above, are often insufficient to define most formal languages completely and additional rules are required to define the syntactic rules of the language more precisely, e.g. these rules are given in prose in OCL 2.0 specification and are extremely language-specific, which means that they have to be implemented manually in addition to automatic syntactic parsing routines. Contextual Analysis can be performed by assigning attributes to productions of the context-free grammar. Each attribute is typically a (type, name) tuple and the resulting CFG becomes an attributed grammar. Effectively, contextual analysis consists of attribute evaluation rules, which specify how to compute values of attributes. Attribute evaluator is a subroutine of the parser, which performs execution of attribute evaluation rules to compute attribute values for a given grammar. Our parser subsystem will be based on a left-to-right depth-first tree walker, which is an algorithm used to perform attribute evaluation in a systematic fashion. An L-attributed grammar is a desired type of grammar, which we obtain, since it enables our algorithm to perform evaluation of all attributes of the grammar with a single traversal of the AST. A particularly relevant grammar which we will use is formally referred to as LALR(1) grammar. Look-Ahead Left-to-Right (LALR) property is desired since it makes the grammar expressive enough to accommodate OCL and results in a parser, which requires the smallest size of parsing tables. LALR is a refinement to constructing Left-to-Right parse tables, and its advantage is due to the use of lookahead sets, as opposed to follow sets, which are typically used for LR grammars. Lookahead sets carry more context-specific information, which is required for attribute evaluation of OCL, and cannot be represented using follow sets.

The relation of syntactic analysis, as described above, to syntactic evaluation subsystem which we implemented in OCL Evaluation Framework is as follows: our subsystem uses syntax parsing, to verify well-formedness of OCL syntax with respect to OCL specification, and contextual analysis to compute attributes, which we use to evaluate context-specific correctness of specified OCL constraints. Hence, our contextual evaluation system is based on contextual analysis but performs manually tailored computations in addition, which means that the two terms (analysis and evaluation) do not refer to the same system.

2.3 Semantic Evaluation in OCL

In this section, we establish the notion of semantic evaluation in the context of the Object Constraint Language and exemplify existing approaches to semantic evaluation in OCL. We also describe and present the background theory for goal-directed theorem provers in propositional logic, as this will become useful in the discussion of our proposed approach to semantic evaluation, which we implemented in the OCL Evaluation Framework.

Semantic evaluation in the context of OCL involves inference of some useful meaning (semantics) from the specified OCL expressions and constraints and verification of whether
that meaning satisfies a pre-determined set of attributes. Suitable attributes for semantic evaluation in the context of OCL are as follows:

- **Satisfiability of Constraints.** Since OCL allows specification of logical constraints, it is possible for one to specify a set of constraints, which are contradictory in themselves. This could either be because they specify numerically and/or logically unsatisfiable situations or because they constrain the underlying UML metamodel to such an extent, that no satisfactory model can be instantiated. It is therefore useful to evaluate the satisfiability of constraints statically in order to determine if the OCL specification itself is contradictory and hence cannot be evaluated successfully against a UML model. There is however a lack of research in this area and many questions still need to be solved. It is questionable whether satisfiability is in general decidable, especially in the case of collection-type constraints.

- There are other kinds of semantic evaluation which can potentially be useful in OCL, such as execution of constraints on a snapshot of the system, runtime evaluation of constraints by the system itself and automated generation of test cases for the system. Due to the fact that there are no current workable implementations of any of these types of semantic evaluation, the latter are discussed in greater detail in Section 7.1 on future work.

In the rest of this section we present the only publicly available implementation of semantic evaluation, the HOL-OCL software, and specify a goal-directed theorem prover in propositional logic, which we will be using in the design of OCL Evaluation Framework.

### 2.3.1 Existing approach: HOL-OCL

HOL-OCL involves Higher-Order Logic (HOL), which we will introduce first.

Propositional logic can be used to express facts using atoms (e.g. a, peter) and logical connectives (and, or, implies). First-order predicate logic extends this system with predicates (e.g. happy(peter)), which take atoms as arguments and quantifiers for-all and exists, e.g. \( \forall x (\text{happy}(x)) \land \exists y (\text{lovesOcl}(y)) \), which can specify facts for “every object (in the situation)”. Second-order logic provides an additional quantifier “for every set of objects (in the situation)”, which increases the strength of expression without addition of any extra predicates. E.g. in second-order logic one can specify \( \forall P \forall x (x \in P \lor \neg (x \in P)) \), i.e. for all sets P and elements x, either x is in P or it is not (law of excluded middle). Second-order logic is a subset of Higher-Order Logic (HOL), which is characterized by so called “super-predicate” symbols, which can take other predicate symbols as arguments in addition to atoms, e.g. \( \text{happy(person(x), time(y))} \) could describe whether a particular person x is happy at some time y, where person and time are predicates. This is a super-predicate of order 2, since it takes predicates of order 1 as arguments and hence could be specified in second-order logic. However, \( \text{circle(oval(shape(x)))} \) super-predicate is of order 3, since it takes a predicate of order 2 as an argument, and is part of the HOL formalism. HOL allows one to evaluate and manipulate expressions, which involve numerical relations, defined using logic notation. A accessible introduction to predicate logics is given in [HOL].

**HOL-OCL** is an interactive proving environment for UML/OCL models which has been developed as a “shallow embedding” into Isabelle, which is a Higher Order Logic (HOL) theorem prover developed at the University of Cambridge. Isabelle allows expression of mathematical formulae and enables theorem proving with these formulae in HOL ([Isabelle]). HOL-OCL’s architecture is illustrated in Figure 2.9.
HOL-OCL uses the following main subsystems:

- **Proof General** is the default user interface to Isabelle, which requires XEmacs to be installed (a syntax editor under UNIX).

- **su4sml**, which is the data repository, used for importing and representing UML/OCL models internally. It supports XMI and is defined using the SML standard, originally developed as an XML based modeling language for services.

- **Datatype Package**, which encodes UML/OCL models into HOL automatically, which can later be processed by Isabelle.

- **HOL-OCL Library** is an extension to Isabelle's core libraries, and it models the theorems needed for expressing the formal semantics of OCL built-in operations and types (Integer, Real, String etc).

- **Theory Morpher**, which an extension to Isabelle's proof procedures, based on rewriting and tableaux techniques. It is needed due to the fact that logical expressions in OCL contain extra semantics, which is not handled by built-in Isabelle procedures, e.g. OCL expressions can involve type OclUndefined, which requires Kleene logic, that comprises three states - false, true and ⊥ (unknown), where, informally, ⊥ is stronger than false but weaker than truth. OCL expressions can refer to "initialisation states" (using @pre) and can simply be assertions, which do not evaluate to any type. HOL-OCL defines a tableux prover to handle examples such as these.

(A more detailed description of HOL-OCL is given in [HOLOCL](#)).

HOL-OCL enables semantic evaluation, such as proving **satisfiability of OCL invariants**, and testing whether **post conditions contradict given invariants**. It also supports all of OCL types and can reason on collection OCL types as well as basic types. But there are significant limitations to implementing a complete self-contained stand-alone package for semantic evaluation using HOL-OCL:
- **Incomplete range of constraints** is supported. HOL-OCL does not provide automated proof facilities for testing satisfiability of constraints against initial and derived value expressions as well as attribute and query operation definitions.

- **Limited extent of automation.** Although the Theory Morpher component of HOL-OCL provides some automation for relatively trivial cases, automation is not guaranteed for non-collection types. In the case of collection types, there are undecidability limitations of the underlying logic itself (no complete first-order logic theorem provers exist, which implies that HOL problems are in general undecidable). But even in the case of decidable examples, HOL-OCL does not provide significant automation, and the user might have to specify additional example-specific theories manually in the Isabelle environment.

- **Lack of Platform Portability.** Isabelle binaries are only provided for Linux and MacOS X, which makes HOL-OCL only compatible with UNIX based systems.

- **Absence of a Direct Integration Method.** HOL-OCL uses the Proof General visual editor for input, which makes providing the editor functionality as-is the only practical way to integrate this environment into other OCL software.

- **Requires Knowledge of HOL Proof Theory.** A software designer constitutes a typical user of OCL, in the context of the model-driven engineering paradigm. A critical requirement of OCL software is therefore to provide an easy-to-use interface to functionality in a way, which requires only the knowledge of UML/OCL modeling from the user. HOL-OCL requires software designers to know Isabelle syntax (which is part of automated proof theory in HOL) to the extent, that is sufficient for understanding proofs and producing them manually, in cases where automation is not possible.

For these reasons, HOL-OCL does not present a readily usable and accessible semantic evaluation system. Given that it constitutes the only current publicly available approach, there is still much to be done just in the context of satisfiability evaluation. In the Design and Specification chapter, we present a subsystem, which provides completely automated semantic evaluation for non-collection types. The system we develop requires no logic background to execute the evaluation of OCL constraints, and basic knowledge of propositional logic to understand the proofs produced.

In the following section, we describe a goal-directed theorem prover in propositional logic, which sets the background to our implementation of semantic analysis in OCL Evaluation Framework.

### 2.3.2 Goal-Directed Theorem Prover for Classical Logic

In the context of classical (propositional) logic, one is interested in designing an automated proof system, which is **sound** (if it proves a theorem, then the latter must be valid) and **complete** (it can prove all possible valid theorems). The general problem can be specified as $P \vdash G$, i.e. attempting to prove some **goal** $G$, given a set of true facts $P$, we refer to as the **knowledge base** (KB). Some automated provers are based on natural deduction (e.g. ANDP and Muscadet), since it is itself sound and complete. Other approaches to automatic theorem provers are based on resolution methods, which are based on translating $P \cup \{\neg G\}$ into conjunctive normal form (CNF) (“conjunctions of disjunctions”) and checking its consistency. E.g. given a problem $a \rightarrow b \land a \vdash b$, we produce $\neg a \lor b \land a$ for the knowledge base P and $\neg b$ for $\neg G$. A resolution-based prover would check consistency of $\{\neg a \lor b, a, \neg b\}$. Resolution of $a \land \neg a$ gives $\{b, \neg b\}$, which gives $\{\bot\}$, i.e. the set is inconsistent and $P \not\vdash G$. Resolution provers have disadvantages in that resolution steps they use to check
for consistency have no immediately clear meaning in relation to the original knowledge base $P$.

Consequently we detail a goal-directed theorem prover for classical logic, which is based on a set of intuitive rules, that are dependent on the type of current goal $G$ and are applied in a fixed order to $P \vdash G$. The aim of these rules is to reduce $P \vdash G$ to $P' \vdash G'$, where $P'$ is simpler than $P$ and $G'$ is simpler than $G$. E.g. reduce showing $P \vdash G_1 \land G_2$ to showing $P \vdash G_1$ and showing $P \vdash G_2$. Our specification is based on [GDProvers].

Goal-directed proof theory is based on a fragment of classical logic $\{\land, \bot, \to\}$ which consists only of conjunction, falsity and implication. Therefore, the first step of an automated proof procedure using goal-directed theory involves translating the given problem $P \vdash G$ into RFC (ready for computation) form. For any well-formed formula (wff), these can be defined recursively as illustrated in Figure 2.10 (recursive steps denoted by $^*$):

Let us exemplify a translation of $(a \lor b) \rightarrow (c \land d)$:

1. $(a \lor b) \rightarrow (c \land d) \implies ((a \lor b) \rightarrow c)^* \land ((a \lor b) \rightarrow d)^*$ by rule (1)

   First sub-tree
   (a) $(a \lor b) \rightarrow c \implies [(a \lor b)]^* \rightarrow c$ by rule (5)
   (b) $a \lor b \implies (a \rightarrow \bot) \rightarrow b$ by rule (8). No further simplification possible, i.e. we have reached RFC. Overall:
   $((a \rightarrow \bot) \rightarrow b) \rightarrow c$.

   Second sub-tree
   (a) $(a \lor b) \rightarrow d \implies [(a \lor b)]^* \rightarrow d$ by rule (5)
   (b) $(a \lor b)$ processed the same way as above. Overall: $((a \rightarrow \bot) \rightarrow b) \rightarrow d$.

2. The two subtrees give the overall final RFC form:
   $((a \rightarrow \bot) \rightarrow b) \rightarrow c \land ((a \rightarrow \bot) \rightarrow b) \rightarrow d$.

Formally, a formula $A$ is said to be in RFC form if it satisfies ONE of the following:

1. $A$ is atomic, or $A$ is $\bot$
2. A is of form \( B \rightarrow C \), where B satisfies (1) and C is of form \( \bigwedge_i A_i \), such that each \( A_i \) is in turn in the RFC form.

Definition 2.2 of [GDProvers] specifies formal simplification rules, which our goal-directed theorem prover should apply in order. We express these in simplified form:

1. **Conjunction Rule**: given \( P \vdash G_1 \land G_2 \), replace the problem by independently showing \( P \vdash G_1 \) and \( P \vdash G_2 \). In natural deduction, this corresponds to \( \land \)-introduction at the end of the proof.

2. **Implication Rule**: given \( P \vdash A \rightarrow B \), simplify to showing \( P, A \vdash B \). This would correspond to applying \( \rightarrow \)-introduction rule in natural deduction (once we have shown that assuming \( P,A \) gives \( B \)).

3. **Atomic Rule** comprises four cases (at this stage, the goal is guaranteed to be atomic).
   
   (a) **Direct Goal**: given \( P \vdash G \), goal \( G \in P \) holds. This is the first base case of a completed proof: we already have the goal, hence the proof succeeds directly.
   
   (b) **Direct Bottom**: given \( P \vdash G \), \( \bot \in P \) holds. This is the second base case, we already have falsity in the KB, hence the proof succeeds for goal \( G \), since \( \bot \rightarrow \) for any wff \( B \) (falsity implies anything).
   
   (c) **Implied Goal**: reduce \( P, A \rightarrow G \vdash G \) to showing \( P, A \rightarrow G \vdash A \). Clearly if we show the antecedent \( A \) of the implication in the knowledge base, then by that very implication (modus ponens in classical logic) we derive the current goal \( G \). Hence, we could step through to showing the antecedent \( A \). The implication is not strictly needed in the new KB, since it does not provide the prover with any extra semantics anymore.
   
   (d) **Implied Bottom**: reduce \( P, A \rightarrow \bot \vdash G \) to showing \( P, A \rightarrow \bot \vdash A \). This rule is reached in case the goal is not implied directly (i.e. atomic sub-rule does not apply) when we could have an implication of falsity in the KB. Similar to the atomic subrule, the implication is not required in the new knowledge base, since we already used its semantics to apply modus ponens.

4. **Restart Rule**: given some state \( P_0 \vdash G_0 \) (\( G_0 \) is atomic) that was previously reached in the proof, and current state \( P \vdash G \), attempt to show \( P \vdash G_0 \). The motivation behind this rule is given by cases where the prover starts with the original problem \( P_0 \vdash G_0 \) and gets to this rule (which means all previous rules did not apply) in the state \( P_1 \vdash G_1 \), where \( G_1 \) is a different goal, which could result from applying rules 2, 3c and 3d. At this point the current knowledge base \( P_1 \) is likely to be different from original \( P_0 \) and contain facts, which can now prove the original goal \( G_0 \). By restarting the proof with the previous goal \( G_0 \), we point the prover to use new semantics of \( P_1 \), and not reproduce the already existing derivation for \( G_0 \). Note that this is the rule of “last resort”, since there are no more rules left to apply after this. Hence if the rule fails, the whole proof fails.

Theorems 2.2 and 2.3 of [GDProvers] establish that this goal-directed system is sound and complete for propositional logic, which ensures that if none of the rules above can be applied, then no valid proof of \( G \) exists given \( P \).

Finally, let us exemplify application of the specified goal-directed calculus for the following problem:
\[
\{ p, p \land q \rightarrow r, (r \rightarrow q) \rightarrow \bot, q \} \vdash \neg r.
\]

**Goal-directed Proof:**
1. Translation to RFC form gives the following initial state:
\{p, p \land q \rightarrow r, (r \land q) \rightarrow \bot, q\} \vdash r \rightarrow \bot

2. Implication Rule. Antecedent \( r \) is added to the KB (it is already there):
\{p, p \land q \rightarrow r, (r \land q) \rightarrow \bot, q, r\} \vdash \bot

3. Atomic Rule (Implied Goal). \( \bot \) is implied by \((r \land q)\) in the KB, hence prove the antecedent (implication is removed from the KB):
\{p, p \land q \rightarrow r, q, r\} \vdash r \land q

4. Conjunction Rule requires the prover to produce a sub-proof for each of the conjuncts.
   First sub-proof
   (a) \{p, p \land q \rightarrow r, q, r\} \vdash r
   (b) Atomic Rule (Direct Goal). \( r \) is already in the KB, hence the sub-proof succeeds.

   Second sub-proof
   (a) \{p, p \land q \rightarrow r, q, r\} \vdash q
   (b) Atomic Rule (Direct Goal). \( q \) is already in the KB, hence the sub-proof succeeds.

5. Both sub-proofs succeeded, hence the Conjunction Rule succeeds and the original theorem is proved.

This example illustrates certain additional features that would be required in an implementation of the goal-directed calculus. The knowledge base needs to defined as a mathematical set, i.e. it should not contain duplicate elements (to improve space complexity) and the order of elements should be undefined. Furthermore, the prover implementation needs to maintain a history of previous atomic goals for the restart rule. Finally, looping needs to be prevented. The simplest way to implement this is to maintain a history of states, so that the same state is not visited twice. There is however, a better solution: in the absence of duplicates in the KB, looping can only occur as a result of applying the restart rule. Therefore, we could modify the calculus to apply the restart rule iff we can further simplify the state, that would be obtained. Given that the prover is to be restarted with some previous atomic goal \( G_0 \), we should only proceed if ANY of the following holds:

- \( G_0 \) is in the current KB (in which case we will succeed in the next step via Atomic Rule (Direct Goal)).
- the current KB contains \( A \rightarrow G_0 \) for some \( A \), in which case we can apply Atomic Rule (Implied Goal) in the next step and obtain a simplified state.

Note that we do not need to check for implied falsity in the current KB, since if such implication existed, it would be “picked up” by the corresponding subrule of the Atomic Rule, and hence we would not get to the restart rule, which we are in. Similar reasoning apply to checking for direct falsity in the current KB.

We have thus completed the specification of goal-directed calculus, which will be extended for design of semantic analysis in our OCL Evaluation Framework.

2.4 Underlying Technology

This section outlines the technology, which we will be using to design, implement and test our OCL Evaluation Framework. We omit obvious details, such as the Java programming language, which is the main language used in our system and Eclipse IDE, which has been used for development of the source code.
2.4.1 Poseidon for UML

We have used Poseidon for UML Professional Edition (see [Poseidon]) version 6.0.2 to develop UML class models, which were employed in the testing of OCL Evaluation Framework. It is a sophisticated environment for generating UML models and diagrams (e.g. class and sequence models, state machines etc). It allows importing UML models in the XMI format, which enables using UML class models with our software. Due to incompatibilities between different versions of XMI and vendor implementations of these versions (see Section 2.1.4), it is critical that Poseidon for UML is used for generating UML models, which are to be used with OCL Evaluation Framework. Using a different version of XMI and/or different UML modeling tools to generate UML models can result in erroneous behavior and unwanted side-effects in our software.

2.4.2 Java Metadata Interface (JMI)

JMI (see [JMIIntro]) comprises a set of interfaces, which can be used to implement a system for managing creation, storage and access of metadata. It is based on the MOF architecture (discussed in Section 2.1.3), designed by OMG. MOF provides a metamodel, which can be used for building UML and OCL models through specifying metadata elements. JMI defines Java interfaces to these elements, and thus enables access of metadata, used to represent these elements. It also caters for exchange and storage of metamodels via XML, because it supports the XMI standard for metadata interchange.

Popular implementations of JMI include the Reference Implementation from Unisys and Sun’s open-source implementation, provided by the NetBeans group, which is known as the NetBeans MDR.

2.4.3 NetBeans Metadata Repository (MDR)

NetBeans MDR (see [NBMDRIntro]) implements JMI to provide a repository for models specified using the MOF metamodel. In the context of this project, we are interested in utilizing NetBeans MDR for storing and accessing UML and OCL metamodels, which are defined in MOF (Metamodels are discussed in Section 2.1.3). Our implementation of OCL Evaluation Framework is based on extending the Dresden OCL Toolkit, which contains definitions of UML and OCL metamodels in MOF, and uses NetBeans MDR to import (via XMI) and access UML class models from the main application.

A critical advantage of NetBeans MDR is that it generates a Java API for accessing loaded metamodels at run-time (of MDR), and hence does not require us to specify our own custom interfaces to the underlying integrated UML/OCL metamodel, which our software generates. The MDR supports the latest XMI standards (versions 1.1 and 1.2) for exporting the contents of the repository and the latest official version of MOF (1.4), which enables OCL Evaluation Framework to support the latest completely specified UML version (1.5, since specification of superstructure of UML 2.0 is incomplete at this time).

2.4.4 SICStus Prolog

SICStus Prolog 3.8 is one of the popular vendor implementation of Prolog, which we have used in this project. Prolog is a powerful programming language developed at the University of Marseille, which is ideal for practical programming in logic. We assume the user is familiar with the Prolog standard, as this knowledge will be required to follow the implementation details of semantic evaluation subsystem, which we present as part of our
OCL Evaluation Framework. We will be using the Jasper interface, provided by SICStus, to implement two-way interactions between our SICStus Prolog runtime system and the main Java application. See SICStus Prolog Documentation [SICStusDocs] for detailed description of provided Prolog syntax and Jasper interface.

2.4.5 Wolfram Mathematica

Wolfram Mathematica (see [WolframMathIntro]) is an integrated environment, which can be used for computation, modeling, simulation, documentation and presentation of scientific data. We will be using Mathematica 6 kernel, which is an interface to basic functionality, for implementing semantic evaluation. Its main advantages, relevant to this project, are:

- computational usability: Mathematica packages different computational routines under one user-level method. Thus we only specify the problem, and Mathematica automatically chooses the most optimal routine for solving it.
- well-defined Java interfaces: Wolfram provides its own definition and implementation of easy-to-use high-level interfaces, which can be used to access Mathematica from Java. We will be using JLink, already integrated with Mathematica, in our implementation of these interactions.

No prior knowledge of the syntax is required from the reader. We will exemplify and describe our limited use of Mathematica in a self-contained manner.

2.4.6 SableCC Parser Generator

SableCC [SableCCIntro] is a compiler compiler. It can automatically generate a parser, as a set of Java packages in source code, given only the lexical (e.g., names of reserved keywords, such as "self" in OCL) and grammatical definition (the structure of expressions) of the underlying language. SableCC can be used to create a framework for the underlying language (which in our case is OCL), such that:

- it enables the parser to build the Abstract Syntax Tree (AST) of the compiled OCL program automatically
- each AST node is typed
- implementation of analysis (tree traversal) of the AST is decoupled from the AST itself, i.e., we can freely extend SableCC’s tree walker classes to gather additional information about the OCL program, which we require for syntactic evaluation. This has been done in Dresden OCL Toolkit, which we extend to implement a more complete syntactic evaluation subsystem.
- storage of analysis information, needed for syntactic evaluation in OCL, is kept within the analysis class itself and not with the nodes of the AST. This prevents any modifications to the structure of the tree and its nodes, and provides a single interface to syntactic evaluation in source code.

We will discuss our usage of SableCC in greater detail when we present the syntactic evaluation subsystem.

This completes our introduction to the technology, which will be used in the development of OCL Evaluation Framework. In the remainder of this chapter, we evaluate state-of-the-art OCL software to further highlight the motivation and significance of features implemented in OCL Evaluation Framework.
2.5 Evaluation of State-of-the-Art OCL Software

In this section we present and evaluate the most popular state-of-the-art OCL software. We do not consider UML modeling tools with OCL support, due to the fact they only provide specifications of OCL constraints, but currently do not support any syntactic (parsing and type checking) and semantic (e.g. satisfiability) evaluation of constraints.

2.5.1 Dresden OCL Toolkit

The Dresden OCL Toolkit (see [DresdenOCL]) is a software platform, which comprises a set of tools at different levels of functionality, which can be categorized as follows (we only consider tools, relevant to OCL):

- **Base Tools**
  - OCLParser 2.0. It is based on the syntax analyser and abstract attribute evaluator interface, generated by an enhanced version of SableCC. The parser is able to provide two types of syntactic evaluation: syntax checking and contextual analysis (referred to as attribute evaluation in the context of SableCC-based implementation). Support for OCL 2.0 syntax is largely incomplete.
  - OCL Base Library 2.0 provides an implementation of built-in OCL types and operations. The architecture is metamodel-based, which accommodates the definition of OCL as a metamodel in MOF, and is compatible with the metamodel-based repository, used by the Toolkit, which we describe below.
  - CodeGenerator generates declarative target code for OCL expressions, which can later be deployed into a definition of a software system. The use of this base tool is exemplified by the OCL22SQL end user tool.

- **End User Tools**
  - OCL Parser GUI provides a simple interface to the OCLParser subsystem: it enables a user to load OCL constraints, and test them against a particular UML class model, which is imported by the user via XMI.
  - OCL22Java uses the CodeGenerator tool at base level in order to produce Java code, that can be used to check specified OCL constraints at run-time.
  - OCL22SQL generates SQL representation of specified OCL invariants together with the underlying UML metamodel, that can be used to verify integrity of the OCL specification. It generates SQL code to create integrity views from a given UML1.5 model and textual OCL invariants.

All of the tools are integrated with NetBeans MDR Repository in order to import and store UML class models, together with which OCL specifications are can be evaluated. Dresden OCL Toolkit provides the broadest support of possible OCL features, but its main disadvantages to end users are related to the fact that the toolkit is developed as a base platform for end-user implementations by third parties, and does not attempt to provide a complete end-user product itself. The software has several critical shortcomings:

- **Incomplete support of OCL 2.0 syntax.** OCLParser base tool provides syntactic parsing and contextual evaluation facilities, but they are largely incomplete:
  - at syntactic parsing level (just checking the syntax of an OCL expression itself, without the associated UML metamodel): name-prefixed contexts (e.g. c:Car),
attribute and operation definitions, allInstances(), oclIsKindOf() and oclIsTypeOf() operations for collection types (e.g. oclIsKindOf(Set(Person))) are not parsed correctly, and result in syntax errors being flagged, although the expressions are well-formed with respect to the OCL 2.0 syntax. In fact, declarations of collection types such as i:Set(Integer) = 1, 2, 3 is not supported at all.

- at contextual evaluation level (checking existence of model elements, referred from OCL, against the underlying UML metamodel): association classes are not supported at all, and qualified navigation of associations is not type evaluated correctly, initial and derived value expression checks are not implemented. Evidently, everything that is not supported at syntactic parsing level, is not evaluated at the contextual level either.

Hence, the OCL parser subsystem requires significant extensions and overhaul of syntactic evaluation before it can provide errorless and complete support of OCL 2.0.

- Lack of usable graphical interface. The GUI provided for the OCLParser base subsystem is experimental, i.e. the interface provided gives access to minimal functionality, the layout is visually inconvenient, names of features are unclear and ambiguous and output for syntactic evaluation errors is almost entirely meaningless to a typical user, who is unaware of the internal structure of the parser. An example of contextual evaluation output is given in Figure 2.11, which illustrates the printout of the stack of Java exceptions that are output. The interface is more suitable for debugging of OCL Toolkit itself and is completely unusable in a software engineering environment.

Figure 2.11: Screenshot of Dresden OCL Parser GUI: Error Message during Contextual Evaluation
The OCL Toolkit, however, has important advantages to OCL developers, such as us, because it is open source and its architecture is modular. In fact, we design our OCL Evaluation Framework by extending the OCLParser subsystem of Dresden.

2.5.2 LCI OCL Environment (OCLE)

OCLE has been developed at BABES-BOLYAI University of Romania, and it represents the most sophisticated attempt at a powerful end-user environment, which involves OCL. It provides a powerful and flexible user interface, which includes a visual interface for creating UML models.

Its key advantage is that it allows to execute OCL constraints on the user UML class model and verify whether the imported UML model conforms to the built-in OCL constraints, which define requirements for the general UML metamodel, i.e. one can check whether the UML model, exported by some other modeling tool is correctly specified in metadata. However, using our OCL constraint test files (which will be illustrated when we describe Testing) helped identify key problems with this software, which relate to incomplete and erroneous support of OCL 2.0 syntax. Even though complete support of OCL 2.0 syntax is promised, in reality lots of valid OCL constructs are not supported and parsed incorrectly:

- at syntactic parsing level: context declarations, which refer association ends and association classes, are not parsed correctly, e.g. declaration context Person::age : Integer throws exceptions; attribute and operation definitions are not supported (similar to Dresden), and well-formed syntax is parsed incorrectly for some collection operations, e.g. \texttt{forAll(p1:Person p2:Person —...)} operation calls with more than one iterator (p1 and p2).

- at contextual evaluation level lots of meaningless errors of the same form, “...Object[Result is undefined]” were flagged when we tried to evaluate OCL specifications (CompanyExample), which are part of official examples, that come with the tool. An example of such evaluation against a user-level UML class model is given in Figure 2.12. A swarm of unclear error messages is produced for a seemingly trivial well-formed OCL specification \texttt{self.employee→select(age > 50)→notEmpty()).}

Hence, overall syntactic evaluation was generally unconvincing and erroneous. Also, the use of visual representation is only supported for UML diagrams defined on-the-fly. When restoring a saved project file, the visual editor for the UML model cannot be opened, and only the tree representation is provided. OCLE also provides code generation facilities for OCL, which we could not test in the absence of any correctly parsed OCL constraint files.

Nevertheless, a key advantage of this tool is an excellent and powerful user interface, which contains a flexible content management system, and allows one to define, view and edit lots of constraints, and selectively test them against the system. A unique feature of this tool is that it also provides visual access to the elements of the generic UML metamodel, although this might not be directly relevant for OCL.

While an effective syntactic evaluation subsystem is practically non-existent, the tool definitely sets a benchmark for flexibility of GUI, which any comprehensive OCL software should be measured against.

36
Figure 2.12: Screenshot of OCLE Debugger: Error Messages during Contextual Evaluation
2.5.3 Octopus

The Octopus tool, developed by Klasse Objecten, runs as a plugin to Eclipse IDE and its source code is publicly available. The developers specify that three key functionalities are provided:

- checking that attributes, operation ends and other model elements of UML are navigated correctly; evaluating OCL expression types in full conformance to OCL 2.0 syntax.
- checking OCL constraints against the UML class model, which can be imported using XMI
- generation of Java code to implement constraints at run-time of the system

However, we have not been able to obtain any sensible syntactic evaluation output. Errors are flagged on trivially well-formed OCL syntax. UML class model is not imported by the XMI module. We have tested Octopus 2.2 on pre-defined examples, which are packaged with it, and used recommended version of Eclipse (3.1), running on recommended platform (we used Windows XP and Vista). Figure 2.13 illustrates a screenshot of Octopus output for the dvdshop example, which ships with the installation binaries.

![Screenshot of Octopus Plugin for Eclipse: Failed XMI import and Incorrect Error Messages](image)

Figure 2.13: Screenshot of Octopus Plugin for Eclipse: Failed XMI import and Incorrect Error Messages

It is hard to make any positive conclusions about Octopus, since we have not been able to exploit any OCL functionality, while following the vendor’s directions for installation and import of examples. Octopus provides proprietary solutions for using it together with Rational Rose UML modeling tool and Poseidon for UML, but they are not publicly available.
2.5.4 Conclusions

We have evaluated the most popular state-of-the-art OCL software, which attempts to provide at least some extent of semantic evaluation. We have not considered the OSLO project, due to its limited scope (it only attempts runtime evaluation of OCL constraints against UML test cases) and error-proneness, since it is based on the Kent OCL Library, which is still under development. It is also worth to note that currently there are significant limitations to widespread use of tools that attempt runtime evaluation, because the formats they support for defining UML instances of the system differ in their implementation across different OCL software.

We have established that none of the OCL packages provide a complete and readily usable syntactic evaluation system, although Dresden OCLParser subsystem represents a good starting point for developing such a system. Recall that none of these software packages attempt any semantic evaluation, such as constraint satisfiability (we have discussed HOL-OCL, the only existing approach to achieve that). OCLE is the only software which provides a flexible and sufficiently powerful interface to OCL.

Hence, there is no ready OCL software that could provide a suitable user interface to creating and evaluating OCL constraints with complete support of OCL 2.0 specification and at least basic attempt of semantic evaluation. This motivates the need for our OCL Evaluation Framework, the design and implementation of which is presented in consequent chapters.
Chapter 3

Specification and Design: Syntactic Evaluation

In this chapter we specify the features of syntactic evaluation, implemented in our OCL Evaluation Framework, and discuss challenging and non-trivial aspects of its implementation and design. We begin with a discussion of the current limitations of OCL 2.0 specification, which gives us an idea of what is possible to design in the context of syntactic evaluation in OCL in general, and sets the tone for a presentation of what we have done in OCL Evaluation Framework.

3.1 Limitations of OCL

Object Constraint Language, as it currently stands, has two main applications:

- it is used by its own developer OMG for defining the UML standard at the metamodel level, e.g. for expressing conditions for valid UML diagrams, also referred to as well-formedness rules (WFRs).
- it is used as a substitute for formal specification languages, such as Z, on the level of application modelling, where it can be used to express invariants, pre- and post-conditions and so on, on the UML class model of the application itself. Although, it still lacks popularity in formal specification, it is positioned as a central language for platform-independent specifications of software components by OMG, and is part of its layered MOF model-driven architecture.

Although, OCL 2.0 has satisfied its previous critics by providing a formal specification of the OCL metamodel, several problems still remain. We discuss these problems (based on [OCL2Eval]), and describe how they have restricted our own specification of functionality in OCL Evaluation Framework:

- **OclMessages.** OCL 2.0 introduces new types of expressions, which allow one to specify that an operation call has been sent to another component of the system over a certain port (precise semantics of the call is not known). A single primitive \( exp \hat{\op} \) \( (params) \) is introduced. The meaning of this call is that it extracts operation calls \( \op \) \( (params) \) from the output message queue \( exp \) and returns them as a collection. There are two major problems with this new type of expressions in OCL:
– since OCL has no way to specify free variables, message expressions can only be used either to specify that a certain operation call has been sent without constraining parameters in any way, or that an operation has been called with explicit parameters (e.g. 5, “peter”). It is impossible to specify constraints on parameters, e.g. one cannot specify an operation call “see(x)”, where x:Real is less than 18.

– it is impossible to express constraints on the order of messages sent. Since only extracting operations with the same name is allowed in one expression, one cannot compare times of occurrence of different operation calls. Furthermore OCL specification does not specify what happens when an operation is called more than once within its pre- and post- condition. E.g. let a:Integer = 3 in adder^add(a) and suppose that two add messages have already been sent, add(2) and add(3). “^” operator extracts only the head of the queue of received calls, so what should be returned: false (but this means add(3) has not been sent) or undefined?

For these reasons, we have not implemented OclMessages in OCL Evaluation Framework. It is certain that OclMessage type will get overhauled significantly in future specifications, which will facilitate its comprehensive integration in our software. Currently, a practical integration into OCL software packages is unclear in the absence of definition of formal semantics of OclMessages in OCL 2.0 specification.

• commonSuperType is an operation defined in OCL used as part of its type inference, i.e. determining types from OCL expressions directly without any explicit type information being given. The OCL specification still remains largely unclear. Consider a situation where each of classifiers A and E extends classifiers B and C. In this case, what is the type of Set {a, e}, where a and e are instances of A and E correspondingly? The OCL specification states that it should be the common supertype of a and e, which in this case could be either B or C. Hence should commonSuperType operation return Set(B) or Set(C)?

For these cases, behavior of commonSuperType is not guaranteed in OCL Evaluation Framework. Furthermore, user could corrupt the type evaluation by exploiting this ambiguity of commonSuperType.

E.g. let s:Set = Set{a, e} in s→asSequence()→first().isOclTypeOf(E) propagates the ambiguity to an operation call “first()”: does it evaluate to type A or E. The chain of type evaluation here is unclear and hence it prevents designing a type evaluator, which follows formal OCL 2.0 specification.

• Concrete Syntax. In OCL 2.0 the grammar for the textual representation of OCL expressions has been significantly reworked to accommodate the newly introduced abstract syntax of OCL, defined using the MOF metamodel. Developers of Dresden OCL Toolkit, which currently contains the most complete parser for OCL, have shown (in [OCL2Eval]) that it is impossible to derive a working parser for OCL 2.0 using the specified grammar as-is without a great deal of guesswork in resolving ambiguities on the way to converting the grammar to LALR (discussed in Section 2.2).

Since Dresden OCLParser already provides LALR grammar (still incomplete) for OCL 2.0, which has taken years of work, we see this as the optimal choice of the underlying subsystem, which we extend to produce an almost complete parser for syntactic evaluation in our OCL Evaluation Framework.
3.2 Specification of the Subsystem

In this section we present the functionality of our syntactic evaluation subsystem. We begin with explicitly defining the notion of syntactic evaluation in the context of OCL Evaluation Framework.

Syntactic evaluation involves two kinds of checks that we perform on the OCL expressions, entered by the user. We will use the following example to illustrate:

```
context Human
inv: Human.allInstances() -> select(age > 50) -> isEmpty()
```

- **Syntax Parsing** of this expression involves checking that the context declaration (context Human) and the associated constraint declaration (inv: ...) conform to the OCL 2.0 grammar. At this stage, the syntactic evaluation subsystem does not care what Human means, and whether allInstances is a defined operation and so on. It only verifies e.g. that a dot can precede allInstances, which in turn can be followed by → in syntax etc. This check requires neither UML nor OCL metamodels.

- **Contextual Evaluation**, on the other hand, requires both metamodels. It verifies several attributes depending on the context:
  
  - **context declarations**: different OCL constraints can be declared only within certain contexts, e.g. invariants are associated with classifiers, initial and derived expressions - with association ends and attributes etc. In our example, context evaluation verifies that classifier Human exists within the corresponding UML class model and checks that an invariant is specified within this context.
  
  - **navigation of model elements**: contextual evaluation must keep track of the context being introduced, e.g. when Human is navigated, the subsystem has to know that it refers to class “Human” and hence verify that operation call allInstances() is allowed on this class. Furthermore, it has to account for the type, which the navigation evaluates to, e.g. navigating an association end with 0..* multiplicity evaluates to Set(Classifier), where Classifier is the type of the associated class in the underlying UML class model.
  
  - **operation signatures and calls**: pre- and post-conditions are defined in the context of operation signatures, e.g. driveTo(x:Real, y:Real) : Boolean, which comprise formal parameter declarations (referred names and types) and a return type. Contextual evaluation checks whether there exists a corresponding operation in the integrated UML/OCL metamodel. In the case of operation calls, e.g. select(age > 50), we check whether the operation exists (either in the UML model or OCL Library) and that the provided arguments evaluate to correct types, e.g. for the shorthand call to “select(condition)” the argument has to evaluate to type Boolean which is satisfied by age>50.
  
  - **source expression types**: in the example above, when select is called contextual evaluation needs to check that the source type of this call is valid, i.e. that it is called on a subtype of Collection. In the example, allInstances() evaluates to Set(Human) which satisfies the required source type of select.
  
  - **constraint features**: invariant constraints must be associated with OCL expressions, which evaluate to type Boolean, attribute and operation definitions must be unique (e.g. cannot specify two definitions on the same attribute), initial and derived value expressions must evaluate to the type of the constrained attribute/association end etc. Contextual evaluation must ensure that the overall type of the whole OCL expression conforms to the required type of the constraint.
In OCL Evaluation Framework we have implemented support for the following OCL features on both, the syntactic parsing and contextual evaluation levels:

- **Built-in OCL Types and Operations.** We have provided support for all parts of the OCL Library covered in the Background Chapter (see Section 2.1.2).

- **All OCL Constraint Types.** OCL Evaluation Framework allows one to specify invariants, pre- and post-conditions, operation bodies, initial and derived value expressions as well as attribute and operation definitions.

- **Loading UML 1.5 Class Models.** Contextual evaluation of constraints is performed against UML class models. Using NetBeans MDR allows OCL Evaluation Framework to support XMI 1.1 and 1.2, and hence load UML 1.5 class models.

- **Generating Integrated UML/OCL Metamodel.** Once the UML class model is loaded, our software generates the integrated metamodel, which consists of the model elements of the UML class model augmented with OCL built-in types and operations. E.g. class *Human* is imported as part of the UML class model, but it is also associated with operations such as `oclIsTypeOf()`, `oclIsKindOf()` etc, which are defined for type *OclAny* in the OCL Library. Since any class extends *OclAny* by OCL 2.0 specification, operations of class *Human* should be augmented OCL operations `oclIsTypeOf()`, `oclIsKindOf()` etc. These augmentations are performed during integration of the loaded UML class model into the integrated UML/OCL metamodel which is displayed in OCL Evaluation Framework. The metamodel is displayed as a tree to the user.

- **Concrete Syntax Tree Visualization.** We have preserved this feature from Dresden OCLParser, which provides the visual representation of the CST of the OCL constraints, defined by the user. We will illustrate this feature when describing the design of our parser subsystem.

- **Semantic Evaluation.** This feature was implemented from scratch, and is discussed in the Design of Semantic Evaluation chapter.

### 3.3 Overall View of the Design

This section describes the design of the parser subsystem and describes some interesting aspects of implementing our extensions to the original Dresden OCLParser. We begin by illustrating an overall view of the subsystem, which we will use to pinpoint exactly which parts of the parser we have modified. The modifications will be described in detail in the subsequent sections.

Areas highlighted in green (also annotated with star) denote parts of the Dresden OCLParser which we have extended to obtain our syntactic evaluation subsystem. OCLParser comprises the actual parser subsystem, which uses the underlying OCLCompiler infrastructure for meta-model access and storage. In the context of syntactic analysis as described in Section 2.2, the parser resides in `tudresden.ocl20.parser` package of the Java source code and consists of the following components:

- **Lexer**, which transforms a character stream, which represents OCL constraints entered by the user, into a token stream

- **LALR Parser**, that converts the resulting token stream into a Concrete Syntax Tree.
• **Attribute Evaluator for OCL 2.0** transforms the CST into an Abstract Syntax Tree, which carries computed attributes for performing contextual evaluation. This is done in accordance with OCL 2.0 specification. In fact, the AST could be thought of as an instance of the OCL metamodel (which can represent all possible CSTs, i.e. sets of OCL expressions), while CST could represent a concrete OCL model (which can represent only the entered OCL expressions).

• **Node Factory**, which is used to create all AST nodes as instances of the JMI-based implementation of OCL 2.0 Abstract Syntax, as defined in the Abstract Syntax chapter of [OCL20Spec]. An interface, which consists of a single method `createNode()` facilitates this functionality.

The parser uses a metadata repository (NetBeans MDR), which represents a centralized location of pre-loaded metamodels, and UML class models loaded by the user as XMI files. This infrastructure constitutes the Dresden OCLCompiler subsystem, which installs three metamodels during the build process of the parser (compilation of the source code for the parser):

• **UML 1.5 metamodel**, with attached OCL 2.0. IUmlOcl interface provides access to the repository via this metamodel.

• **MOF 1.4 generic metamodel**, with attached OCL 2.0. IMofOcl is an interface to this metamodel, but is currently not used, since MOF 1.4 is not supported by the parser yet. This means that OCLParser only accepts UML class models defined using the UML metamodel. Definitions which directly use MOF metamodel for class models, or model the UML metamodel itself using MOF (this is done in OCLE software, evaluated in Section 2.5), cannot currently be imported.
• **Common OCL metamodel**, which describes the abstract syntax of OCL 2.0 irrespective of the underlying model (UML or MOF). It provides a model-independent access to the repository. Currently its advantages are not fully realized due to the fact that MOF models are not supported, hence solely IUniOcl interface could be used for repository access.

Figure 3.2 illustrates UML Activity Diagram for the semi-automatic construction process of the Dresden OCLParser. We have significantly extended the subsystem in areas, which are marked with starts. Sections marked with “manual” indicate that the corresponding activities were performed manually, while all the other sections were automatically generated.

We have provided three main extensions to the original Dresden OCLParser subsystem (SableCC has been described in Section 2.4):

• **Modification of the SableCC extended grammar for OCL**: we have modified some of the production rules and added some tokens and new productions to the grammar, to facilitate an almost complete support of OCL 2.0 syntax on syntactic parsing level. We detail some of these in the subsequent sections.

• **Extension and addition of attribute evaluation rules**: we have added new rules to compute attribute values for the new productions that we added to the grammar. We have also modified already implemented attribute evaluation rules to facilitate correct contextual evaluation of OCL constraints.

• **Adding entry hooks to semantic evaluation subsystem**: our semantic evaluation subsystem has been implemented in a way which satisfies our AST traversal algorithm (tree walker), since all the attributes, necessary for semantic evaluation, are computed within a single traversal of the AST together with all the other attributes. We have manually inserted **code hooks** in attribute computation subroutines, which allow our semantic evaluation subsystem to generate the required semantic form for constraints and collect them for further processing during AST traversal. This is described in greater detail in the Design of Semantic Evaluation chapter.

These extensions have been implemented over several iterations of the construction process, illustrated in Figure 3.2. The latter can be described by the following main stages:

1. **Generating Extended SableCC**. We have already mentioned that SableCC is ideal for generating lexers and parsers, however, it is not capable for generating attribute evaluators. Since it is open-source, developers of Dresden have integrated a facility for generating attribute evaluator skeletons (abstract classes in Java source code). The extended SableCC grammar is fed as input to the original SableCC compiler compiler. It generates complete lexer and parser for the now extended grammar together with the already mentioned abstract base class (**LAttrEvalAdapter** in source code), which serves as an interface to be implemented by attribute evaluators. SableCC also generates some tree walker classes (part of the analysis package, as discussed in Section 2.4), which implement a left-to-right depth-first traversal of the AST.

2. **Implementing the Attribute Evaluator**. A concrete implementation of the skeleton is developed manually (in class **LAttrAstGenerator** in source code), which comprises all the necessary attribute evaluation rules. The latter are implemented by means of “computeHeritage...” and “computeAstFor...” methods, which computed Inherited Attributes (IA) and Synthesized Attributes (SA) respectively. These are described...
Figure 3.2: Construction of Extended Dresden OCLParser (Adapted from [DresdenParser])
in greater detail in the subsequent section. Note that due to our extensions, these methods also contain hooks to the semantic evaluation subsystem.

3. **Compiling the Parser.** This process corresponds to compiling java source files into class files, which are then packed into a JAR file, representing the complete parser subsystem (this process is automatized by using Ant builder). Note that due to our extensions, this JAR file also includes the semantic evaluation subsystem and our graphical interface, and hence contains complete implementation of our OCL Evaluation Framework.

![Figure 3.3: Access of Meta-model Repository with Extensions](image)

Figure 3.3 illustrates the way interactions occur between the parser and the meta-model repository. The user loads the UML using an XMI file through the Graphical User-Interface, which we have designed for OCL Evaluation Framework. The XMI file is loaded into the repository using the `IRepository` interface. `NodeFactory` is used to create AST nodes using interface `ICommon` to instantiate these nodes in the repository (which provides implementation of the Abstract Syntax of OCL 2.0). `NodeFactory` is accessed via concrete attribute evaluator `LAttrAstGenerator`, which implements the abstract skeleton `LAttrEvalAdapter`, generated by our extended SableCC.

### 3.4 Implementing Parser Extensions

This section describes in detail the extensions to the Dresden OCLParser subsystem, which we have implemented. We begin by considering the way in which inherited and synthesized attributes are computed in the AST, and consequently present the implementation of our extensions with reference to the attribute evaluator and SableCC input grammar.
3.4.1 Inherited and Synthesized Attributes

In this subsection, we illustrate the general procedure, which we employ to collect information, necessary for computing attributes during AST traversal.

Consider the following example:

context Human
  inv: self.age > 0

Our contextual evaluation of this example must, amongst other things, verify that `self.age` is a call to an existing attribute `age`. But what does `self` refer to? In another words, when computing attributes to perform this evaluation, we somehow need to ensure that when our attribute evaluator traverses the AST node for `self`, it can access the context of the whole expression in order to infer that `self` refers to an instance of `Human`. This is done by means of computing and passing Inherited and Synthesized Attributes during AST traversal, as illustrated in Figure 3.4.

Recall, that the AST traversal follows a left-to-right depth-first order. Before attribute evaluation is performed at each node, the latter obtains a set of inherited attributes (IA) from its parent in the tree. In our example, node `self` would receive a set of IA, which would include context information, which enables attribute evaluator to determine the context inside `self` and perform the necessary computation of attributes. Now suppose that we are at node `, which corresponds to “>” in the AST. As part of contextual evaluation at this node, our attribute evaluator needs to verify that both operands (`age` and `0`) are of allowed type (Integer in this case). This example motivates the need for synthesized attributes (SA): `age` needs to let its parent (“>”) know that its type is Integer. This is implemented by computing the type of `age` as an SA when our attribute evaluator is at node `age`, and the set of SA is then passed to the parent of the AST node, which is “>” in our example. This
is illustrated for a general case in Figure 3.4. In between entering and exiting an AST node, our attribute evaluator may call other methods to compute auxiliary or other intermediate attributes, which might be required. On exit, the set of all computed SA is passed on.

Each inherited attribute is not passed as a single unit, rather all of IA are passed in a single container, represented by the *Heritage* class in source code. SA are not modeled separately either, and are either passed as a single item, if only one SA is returned, or packed in a single container if more than one SA is returned. Class Heritage contains an associated *Environment* class, which comprises all the variables and identifiers that can be viewed from the current scope. For our example above, at AST node for age, our attribute evaluator would check whether age is a valid identifier from the current scope by attempting to look it up in the Environment class. The latter is retrieved from a set of IA, represented by the Heritage class, by calling Heritage.getEnv() which returns the associated instance of Environment (this class is part of the official OCL 2.0 specification of Abstract Syntax).

We will provide a complete description of how attribute values are computed, and hence how contextual evaluation is performed for logical negation in OCL. We abstract away from automatically generated implementation, such as the tree-walker, generated by SableCC and concentrate on the mapping between the OCL grammar, fed as input to SableCC, and its corresponding attribute evaluation rules, implemented in *LAttrAstGenerator* (our attribute evaluator class). Assume the following example:

```context Human
inv: not self.married  -- suppose married is a
  -- boolean attribute of Human
```

Here a negation, which is a unary operator, that takes a single argument of type Boolean, is used to assert that all Humans in our example are not married. If we look at the relevant extract of the SableCC grammar, it is given by:

**Tokens**

... 
not = 'not';
...

**Productions**

... 
unary_exp_cs <OclExpression> =
  unary_op <OperationCallExp>
    [operator]:unary_op
    [operand]:postfix_exp_cs
  //comment - SA computed here, before the node is exited

  unary_op <String> =
    minus minus #chain
    | not not #chain
    ;
...

As can be seen, this production handles the unary minus operator (e.g. -5) as well as negation. Names in angle brackets (e.g. `< OclExpression >`) specify the type of the AST node, which will be created for the production. In our example, *unary_exp_cs* will have an AST node *OclExpression* (which is a supertype of all expressions in OCL, which we mentioned when discussing the Abstract Syntax of OCL). A production can have many alternatives (in our example, there is only one), hence each production can have its own AST
type, e.g. `OperationCallExp` is the AST node type for alternative `unary_op` of production `unary_exp_cs`. Hence, `OperationCallExp` must be a subtype of `OclExpression`. Once the AST node for `unary_exp_cs` is processed, its type gets upcasted to abstract `OclExpression` from concrete `OperationCallExp`. Production alternative `unary_op` is expressed using two other productions, `unary_op` (which is specified below `unary_exp_cs`) and `postfix_exp_cs` which we will not discuss in the context of this example. Note that `operator` in square brackets indicates that the AST node for `unary_op` will be tagged with the name “operator” (and similarly for `operand`) at the time when our attribute evaluator computes SA for the AST node, corresponding to `unary_exp_cs` production. `unary_op` production is used to parse the name of the operator, which is “not” for our example (hence production alternative `not` will be followed). Note that `not` is a reference to a token, which in turn defines exactly what negation symbol looks like (string “not”) in the input text stream. In this case the attribute evaluation is trivial: the operator name “not” is directly passed on as a String (hence “< String >” AST node type). Keywords, which begin with # (e.g. `#chain` in this case) are special control instructions to the SableCC generator. Note that `unary_op` production alternatives (`not` and `minus`) do not have their own AST node types, in contrast to `unary_op` alternative of `unary_exp_cs`. This is due to the fact that `#chain` specifies these unnamed AST node types to be the AST type of their parent production `unary_op` automatically, which is String. `#chain` is useful for so-called chain rules, which are productions with alternatives, which are just different syntactic names for the same semantic concept. Here `minus` and `not` are just “syntactic sugar” for a single semantic concept, operator name, which can be modeled by type `String` in the AST.

We will now illustrate how synthesized attributes are computed in Java source code by our attribute evaluator LAttrAstGenerator, before the AST node for production `unary_exp_cs` (alternative `unary_op`) is exited:

```java
public OperationCallExp computeAstFor_AUnaryOpUnaryExpCs(
    OperationCallExp myAst, Heritage nodeHrtgCopy,
    String astOperator, OclExpression astOperand
) throws AttrEvalException {
    try {
        myAst.setSource(astOperand); // attach source expr to AST node
        myAst.getArguments().clear();
        // calling an instance of TypeEvaluator to infer operand type and
        // setting it as source type
        Classifier opType = typeEval.getType(astOperand);
        myAst.setSrcType(opType);

        Operation op = opType.lookupOperation(astOperator,
            Collections.EMPTY_LIST);

        if (op == null) { // op not found
            // throw attribute evaluation error as exception
            myAst.setReferredOperation(op); // attach op to the AST node
        }
    } catch (WellFormednessException wfe) {
        // evaluation of types by typeEval may result in errors
        // indicate type evaluation error (well-formedness violated)
    }
    ... 
    // semantic evaluation hooks (discussed later)
}
```
computeAstFor (line 1) names methods, which compute the AST node and associated synthesized attributes. “AUnaryOpUnaryExpCs” names the corresponding production alternative (A stands for alternative) unary_op of production unary_exp_cs which we defined above. Hence this method computes SA as the AST node for unary operator is traversed (recall, that we are dealing with negation in our example). As illustrated in Figure 3.3, this method is called when children nodes, corresponding to productions unary_op and postfix_exp_cs have already been traversed and supplied us with their sets of SA. We can access these SAs via instance of Heritage “nodeHrtgCopy” passed as one of the arguments to our method. “myAst” (note that its type matches the specified type of unary_op alternative in the grammar above) is the node which we have to pass on to resume tree-walking once attribute evaluation is finished for this node. “astOperator” is the AST node we received from the unary_op production, and it should be equal to String “not” for our negation example. “astOperand” is the argument OCL expression, which should be an attribute call self.married in our example. Our attribute evaluator performs two main tasks in this method:

- **attaching attributes to the AST node**, which are required for future attribute evaluation. We attach the source OCL expression (self.married) to the AST node on line 5, set the source type of this expression on line 9, and attach the operation, referred by the AST node, on line 18. These are all elements of the Abstract Syntax of OCL 2.0, some of which we touched upon in our discussion of the OCL metamodel (Section 2.1.3).

- **evaluating relevant attributes for the AST node**, to check that the latter satisfies all of OCL contextual requirements. Our LAttrAstGenerator is associated with an instance of TypeEvaluator, which is used to infer types, in cases where they cannot be inferred directly. In our negation example, “self.married” is an attribute call, but in general the operand could be any OCL expression. The job of the TypeEvaluator instance is to infer the type of the operand expression, which should be Boolean for our example. But how do we check that the operand type should be Boolean for a negation, in the code given above (e.g. how do we pick up that not 5 is contextually incorrect?) This is done implicitly by a call to lookupOperation for our instance of type Boolean on line 11. This exemplifies interactions which occur between the attribute evaluator and the meta-model repository, that we have illustrated schematically in Figure 3.3. Any type is a classifier in the MOF, and is thus modeled by an instance of Classifier implementation class (which resides in package tudresden.ocl20.core.jmi.uml15.impl.core in source code). The latter implements an abstract JMI-based interface Classifier (which resides in package tudresden.ocl20.core.jmi.uml15.core for interfaces, which are part of ICommon and are automatically generated by the repository). The Classifier implementation is instantiated for each type in the repository, which explains why it also represents the Boolean type (line 11). lookupOperation is a method defined for all types (including UML types), which looks up operations defined within that type. E.g. in our Car World (Figure 2.2), type Car contains operation getTopSpeed(), which our attribute evaluator could lookup by via a call to Car.lookupOperation(“getTopSpeed”, Collections.EMPTY_LIST) instead (line 11). Back to our negation example, lookupOperation(“not”, ..) succeeds for type Boolean but will not succeed for, say, not 5 (since 5 is of type Integer), which explains how contextual evaluation is performed in this case.
It is important to note that AST nodes generated by our parser, such as `OclExpression`, `OperationCallExp` etc, are instances of classes of the OCL metamodel, we discussed in the Background chapter (Section 2.1.3) and as a result they conform to the abstract syntax of OCL 2.0 specification.

In the following sections, we describe some interesting aspects of extensions and corrections, which we have implemented for syntactic evaluation in OCL Evaluation Framework.

### 3.4.2 Adding Support for Context Naming

Trying to parse the following OCL constraint on the latest version of Dresden OCLParser, 1.2, gives an error:

```ocl
context c:Car
  inv: c.model <> ''
```

The problem occurs because named contexts are not supported on the syntactic parsing level and therefore not on the contextual evaluation level.

#### Syntactic Parsing Extensions

The relevant parts of the original OCL grammar, which is fed into SableCC, are as follows. *Grammar 1.1:*

**Tokens**

... 

dbl_colon = '::';

...

**Productions**

...

context_declaration_cs <OclContextDeclaration> =
  attr_or_assoc <OclAttrOrAssocContextDecl> context
  [context_name]:path_name_cs ... 

  | classifier <OclClassifierContextDecl> context
  [context_name]:path_name_cs ... 

  | operation <OclOperationContextDecl> context
  [context_name]:path_name_cs ... 

; 

path_name_cs <List> = [qualifier]:path_name_head_cs*
  [name]:identifier_cs #nocreate;

path_name_head_cs <String> = identifier_cs dbl_colon #chain;

...

```
context_declaration_cs is a production which specifies the syntax of any context declaration in OCL. Three alternatives are possible: classifier declarations (e.g. context Human, as for invariants), attribute/association end declarations (for initial and derived value expressions, e.g. context Person::age : Integer declares an attribute context) and operation declarations (for pre- and post- conditions, e.g. context Human::getAge() : Integer declares a context for getter operation for attribute age). The path to the constrained element is
defined to be the first part of the declaration for all alternatives (named \texttt{context\_name}) and is specified by the \texttt{path\_name\_cs} production. This production specifies that a path can consist of 0 or more (indicated by \texttt{*}) “path prefixes” of the form \texttt{name::} which always end with \texttt{name}. Let us exemplify with some syntax:

- \texttt{carworld::Person::age} conforms to the definition. There are two \texttt{path\_name\_head\_cs} production instances: \texttt{carworld::} and \texttt{Person::}, while age is matched as \texttt{[name]:identifier\_cs} of \texttt{path\_name\_cs} production.

- \texttt{age} also conforms to the definition, since \texttt{*} indicates that 0 production instances is allowed as well. In this example age is matched with a production named “name” in \texttt{path\_name\_cs}.

- \texttt{c:Car}, however, does NOT satisfy the definition, which is the reason for parse errors produced by original Dresden OCLParser. It cannot be matched to “qualifier” since the latter involves double colons (\texttt{::}), and cannot be matched to “name” since the latter is just a name string (i.e. cannot contain control characters, dots, colons and semi-colons).

We modify the \texttt{context\_declaration\_cs} production as follows. \textit{Grammar 1.2}:

\begin{verbatim}
context\_declaration\_cs <OclContextDeclaration> =
  attr\_or\_assoc <OclAttrOrAssocContextDecl> context
  [context\_tag]:name\_prefix\_cs? [context\_name]:path\_name\_alt\_cs
    colon [type]:type\_specifier ...
  | classifier <OclClassifierContextDecl> context
  [context\_tag]:name\_prefix\_cs? [context\_name]:path\_name\_cs
    [constraints]:classifier\_constraint\_cs+ #customheritage

  | operation <OclOperationContextDecl> context
  [context\_tag]:name\_prefix\_cs? [context\_name]:path\_name\_cs ...

// NEW PRODUCTION
name\_prefix\_cs <String> = [name]:simple\_name colon;
// NEW PRODUCTION
path\_name\_alt\_cs <List> = [qualifier]:path\_name\_head\_cs+
  [name]:identifier\_cs #nocrcreate;
\end{verbatim}

We have modified the original production alternatives for context declaration by inserting our own production \texttt{name\_prefix\_cs}. It simply specifies that original context declarations can now be optionally (specified by “?”, which indicates 0..1 multiplicity of production) preceded by \texttt{name\_prefix} which will render declarations such as \texttt{c:Car} valid and will not contradict already allowed declarations such as \texttt{carworld:car} (in which case, the multiplicity of \texttt{name\_prefix\_cs} production is 0). Note that \{\texttt{attr\_or\_assoc}\} alternative for attribute/association end context declaration has been modified further and involves a new production \texttt{path\_name\_alt\_cs}. The reason for this is that declarations such as \texttt{c:Car} will result in a parser conflict if we used the original \texttt{path\_name\_cs} production (as defined in \textit{Grammar 1.1} above) in place of \texttt{path\_name\_alt\_cs}. The parser will get stuck when trying to decide on what to match with the syntax next, when inside production alternative \{\texttt{attr\_or\_assoc}\} (each path through the grammar must be unique, in that the same syntactic input must match at most one chain of productions; otherwise shift/reduce errors result):
• **OPTION 1.** \(c:Car\) could be matched as \([\text{context_tag}]\):name_prefix_cs\) (matching \(c:\) followed by \([\text{context_name}]\):path_name_cs\) (matching \(Car\)).

• **OPTION 2.** Because of “?” the parser can omit \([\text{context_tag}]\):name_prefix_cs\) and match \(c\) to \([\text{context_name}]\):path_name_cs\) “?” to colon and \(Car\) to \([\text{type}]\):type_specifier\) (which is a production, defined in the original grammar, which specifies valid syntax for types, e.g. Human, Integer, Real etc). This matching path exhibits the incorrect definition of the alternative for attributes/association ends in the first place. Even if it does not support context naming, it should not allow \(c:Car\) as a declaration of attribute/association end, even if “c” was an actual attribute/association end name, because the latter must always be prefixed with the name of its parent classifier, e.g. \(Car::age\) ...

To motivate the solution, let us consider declaration “context \(Car::age : Integer\)”. As just mentioned, attribute/association end declaration *always* requires the name of constrained attribute/association end to be prefixed with the name of its parent classifier, hence the definition of \(\text{path_name_cs}\) in Grammar 1.1 does not suffice for alternative \{attr_or_assoc\}, because “*” implies we could have 0 “pathname heads”, i.e. we could “get away” with specifying only “context age : Integer” instead of “context \(Car::age : Integer\)”. Hence only for alternative \{attr_or_assoc\}, we specify a different production for matching paths to constrained elements, given in Grammar 1.2 as \(\text{path_name_alt_cs}\). It is almost equivalent, except that it uses “+”, which gives semantics of 1..n, i.e. we force attribute/association end name to have at least one “name::” prefix, i.e. it must have at least one parent.

We have now resolved the conflict, since the parser cannot use OPTION 2 (stated above) to match “\(c:Car\)” (OPTION 2 requires at least one “::”-prefix, e.g. something::\(c:Car\)) and is therefore confined to using OPTION 1, i.e. we have produced a unique path through the grammar as required.

This completes implementation on the *syntactic parsing level*.

**Contextual Evaluation Extensions**

Consider the following OCL constraint (the associated UML model is given in Figure 2.2):

```ocl
class Person{
  inv: pers.ownedCar->select(c:Car | c.model = 'Carrera GT' and c.producedBy.name = 'Porsche')->notEmpty()
}
```

It expresses the supposed property of our Car World that every person owns at least one “Carrera GT” car, made by “Porsche”. We retrieve a set of all associated cars for a given person (navigating pers.ownedCar gives Set(Car)) and select all cars, which have their model name equal to “Carrera GT” and are produced by “Porsche”). Note that \(c.producedBy\) navigates association end and should in general return Set(Producer), but since the multiplicity at that end is exactly 1, OCL specification allows using attribute-like dot notation for navigating this association end. \(notEmpty()\) asserts that at least one Porsche Carrera GT must be owned.

The problem occurs during contextual evaluation of pers.ownedCar: how does our attribute evaluator (LAttrAstGenerator) know that pers is another name for an instance of type Person? In essence, the semantics of “pers” are exactly the same as those of “self” as we can always write \(self.ownedCar\). We insert this information into the attribute evaluation process as follows:
WritableEnvironment expressionEnv = parentEnv.nestedEnvironment();

if (astNamePrefix != null) { // context named
    try {
        VariableDeclaration vardecl = createVariableDeclaration(
            contextualClassifier, astNamePrefix.getName());
        expressionEnv.addElement(astNamePrefix.getName(),
            vardecl, true);
    } catch ...
}

Firstly, we obtain an instance of Environment (contains all visible identifiers, e.g. attributes, operations etc for our node), which we can write to (on line 1; note that we can only write to our own environment - a child node cannot modify a parent’s environment). We then attach a variable declaration to the environment (line 5), which is specified by its name (which would be “pers” for our OCL example above) and associated type (in our example, this is the named contextual classifier, “Person”). “astNamePrefix” denotes the String name of the context, given by user. It should already be available to us as it is returned by our child. In Grammar 1.2 above, we have specified alternative {classifier} for production context_declaration_cs completely.

The specification of classifier context (e.g. context Person) is followed by one or more constraints (indicated by +) on this classifier. “#customheritage” control flag indicates that the preceding element in the grammar (“constraints” in our example) should inherit attributes, which will be computed by a custom code hook (in accordance with a general illustration we have provided in Figure 3.4), which we implement in LAttrAstGenerator. “#customheritage” is flagged for other alternatives of production context_declaration_cs in a similar way. Hence we insert the code, which we have just provided, into the three methods (one for each alternative), which compute IA for “constraints”. All the nodes and children which represent constraints, defined within the declared context, will inherit attributes, computed by these methods, and hence will also inherit the semantics of “pers” which we have provide using the code above. We thus insert this code into the following methods:

- `insideAAttrOrAssocContextDeclarationCs_computeHeritageFor_Constraints(...`, which computes IA for constraints defined within attribute/association end context declaration (e.g. context Car::age : Integer)

- `insideAOperationContextDeclarationCs_computeHeritageFor_Constraints(...);` similar method, which computes IA for constraints within operation context (e.g. context Car::getTopSpeed() : Real)

- `insideAClassifierContextDeclarationCs_computeHeritageFor_Constraints(...);` equivalent specification for classifier constraints, i.e. invariants (e.g. context Human).

Now, when `ownedCar` is evaluated in `pers.ownedCar` from our OCL example, identifier “pers” will be looked up in the Environment successfully by our attribute evaluator. This is done in a `computeAstFor...` method for computing synthesized attributes for the AST node `AssociationEndCallExp`, which will represent a call to association end `ownedCar`. It also corresponds to a node with the same name in the OCL metamodel (discussed in Section 2.1.3). The implementation is now complete on the contextual evaluation level as well.
### 3.4.3 Implementing allInstances() Operation

allInstances() operation introduces a powerful framework of specification on collection types, because it allows manipulating object instances much like records in a database:

```ocl
context Human
inv: Human.allInstances()->reject(h:Human | h.parents->isEmpty())
->size() == Human.allInstances().size()
```

This OCL invariant specifies that all Humans must have parents: we take all instances of “Human”, and produce a subset which results when humans, which have no parents (this could be modeled by navigating association end parents, with multiplicity 0..2), are removed (this is done via `reject` operation). We then assert that the size of this subset is the same as the size of the set of all humans, which can only hold iff the subset, which results from removing “parent-less” humans, is empty (H represents the set of all instances of “Human” in the system):

\[
\|H\|_\text{parents} = 0 = \|H\| \Rightarrow \|H \cap H'\|_{\text{parents}} = 0 = \|H\| \Rightarrow H'_{\text{parents} = 0} = H_{\text{parents} > 0} \quad H = \emptyset.
\]

#### Syntactic Parsing Extensions

allInstances() is not supported in Dresden OCL and produces errors on contextual evaluation level. The reason for that is that the original attribute evaluator does not take into account the fact that allInstances() is an unusual operation, as it is called on the class itself (“Human”) and not on an instance of that class, which is typical of other operations (e.g. `self.operation()`). Also, allInstances() is parsed as a generic identifier name, while we would like the parser to formally recognise that allInstances() is called.

We modify the OCL grammar as follows. *Grammar 2.1:*

```
Tokens
all_instances = 'allInstances';
...
```

```
Productions
identifier_cs <String> =
  simple simple_name #chain
  | iterate iterate #chain
  | iterator_name iterator_name_cs #chain
  // all instances token added explicitly
  | all_instances T.all_instances #chain
; ... 
```

We have inserted the allInstances token explicitly as an extra alternative of production `identifier_cs`, which completes formal definition on the syntax parsing level.

#### Contextual Evaluation Extensions

Interesting extensions occur on the contextual evaluation level, as we need to modify the way navigation calls are processed (we abstract from implementation on the grammar level):

1. navigation calls in grammar are processed using production `path_name_cs`, which can occur in two contexts, which we will refer to as `primary` and `postfix`.  

56
2. e.g. for context Person inv: self.ownedCar¿... the primary context is self (which results in AST node VariableExp being created by attribute evaluation, since self refers to the underlying context, also known as contextual classifier). The postfix context is ownedCar, which results in AssociationEndCallExp node being created in the AST in accordance with the OCL metamodel (ownedCar is an association end).

3. e.g. for context Person inv: age¿0 the primary context is age, and there is no postfix context, i.e. AttributeCallExp results in the AST. In general, variable expressions, attribute, association end, association class calls can all occur in the primary context.

4. Human.allInstances() results in a problem, because Human is a classifier in the primary context, and is not dealt with by the attribute evaluator, which gives contextual evaluation errors. Furthermore, OCL metamodel does not specify a class for creating an AST node ClassifierCallExp, which would be a logical equivalent of say AST node AttributeCallExp for attributes.

5. note that production path_name_cs is used in conjunction with property_call_parameters_cs for processing operation calls (e.g. allInstances()), with the latter production being needed for processing arguments to the operation (empty list in the case of allInstances()).

Hence, we face two challenges: finding a way to represent the primary context Human with a suitable node in the AST (due to item 3), and implementing a special context evaluation routine, when processing allInstances (i.e. modifying method for computing SA for productions in item 4).

The relevant parts of the method for verifying primary contexts in our attribute evaluator are given below:

```java
public OclExpression computeAstFor_APathTimePropertyCallExpCs_inContextPropertyPrimary( ... ) {

    // lookup variable declaration, e.g. self, named contexts
    ModelElement me = ne.getReferredElement();
    if (me instanceof VariableDeclaration) {
        // create&return VariableExp AST node
    }

    // lookup attribute and its source in the environment
    // e.g. for context Person inv: age...
    // attribute is “age” and its source is classifier Person
    // create&return AttributeCallExp AST node

    // similar for association classes (with source)
    // and association ends (with source)

    // EXTENSION:
    // no other rules to try, check for classifier here

    // e.g. finds specified class in the current package
    Classifier clr = nodeHrtgCopy.getCurrentPackage().
    findClassifier(astName);
    if (clr != null) { // classifier found
```
26 ... 
27     OclExpression oclexp = (OclExpression) factory
28         .createNode("AttributeCallExp");
29     oclexp.setType(clr);
30     return oclexp;
31 }
32 // nothing left to try, throw evaluation error...
33 }

We insert an additional rule for classifiers, if previous rules (for attributes, association ends, variables and association classes) were not matched. This is a method for computing SA, hence we need to generate an AST node to pass synthesized attributes which we compute. As stated in point 3 above, OCL metamodel does not provide an explicit node type for this (e.g. ClassifierCallExp). We “steal” the AST node for AttributeCallExp (used for attribute calls; on line 27), and use it to pass the classifier by attaching it as a type (line 29) of an attribute. This is a somewhat “dirty” method, but it cannot be improved until the OCL metamodel is revised in the OCL 2.0 specification.

On attribute evaluation level, handling of operation calls is done by method for computing SA called computeAstFor_AArgListPropertyCallExpCs(...), which handles both the primary context of operation calls (e.g. context Car inv: getTopSpeed());0) and the postfix context (as in our example Human.allInstances()). We check that the type of the source expression, our “stolen” AttributeCallExp (we have already set the type as classifier Human), is in fact a classifier if the operation name is “allInstances” (since the source type is only allowed to be a classifier for allInstances()).

This completes the type checking procedure of allInstances. It is already specified in the OCL library of operations, and has a return type Set(OclAny). Since any classifier is a subtype of OclAny, the definition does not violate the OCL 2.0 specification.

3.4.4 Other Extensions

In this subsection, we outline interesting parts of other extensions to syntactic evaluation, which we have implemented.

Attribute and Operation Definitions

The original attribute evaluator of Dresden does not support attribute and operation definitions, e.g. declarations such as context Person def attr married : Boolean = true. We have modified the relevant production (defined_entity_decl_cs) in the OCL grammar to support “attr” keyword for specifying attributes and “oper” keyword for specifying operations.

On contextual evaluation level, we have modified corresponding methods for computing synthesized attributes. Since definitions represents assertions, rather than constraints, and hence do not evaluate to any value, the only attribute evaluation procedure required is inserting the newly defined attribute/operation into the integrated UML/UML metamodel. Defined attributes/operations need to be inserted within the classifier, which they are associated with. This classifier is in turn the contextual classifier (e.g. Person) which determines which class in the model contains the newly defined attribute/operation. Hence we modify JMI-based interface “Classifier” (which is part of the ICommon set of interfaces for accessing the repository, illustrated in Figure 3.3) and its corresponding implementation “Classifier-Impl” (part of core.jmi.uml15.impl package) with createOperation() and createAttribute()
methods, which implement the actual insertion of new definitions into the MOF-based metamodel.

Then in our attribute evaluator, methods for processing corresponding AST nodes can just retrieve the contextual classifier via Heritage (inherited attributes) and then call the corresponding method (createAttribute()/createOperation()) on that classifier to insert the new OCL definition:

```java
Classifier ctx = hrtgInstance.getContextualClassifier();
ctx.createAttribute(attrName, attrType);
```

With operations, things are a little more complicated, since we need to generate the list of formal parameter signatures, which needs to be passed to createOperation(). For simplicity, we omit full details of this call.

**Specifying Void in OCL**

The problem with the original OCLParser by Dresden is that it chooses to represent void type as a Java value `null` (e.g. `setTopSpeed()` is a setter operation, which returns void, i.e. type OclVoid in OCL). Thus when we attempt to define pre and post conditions for operation, such as `setTopSpeed()` in Figure 2.2, attribute evaluation produces errors, because attribute evaluator “thinks” that we have not specified the return type of `setTopSpeed` (attribute evaluator also uses `null` to represent unspecified attributes), whilst in reality it is automatically defined as `null` AST node by the attribute evaluator itself.

We modify the corresponding attribute evaluation method (computing SA) to retrieve type `OclVoid` from the OCL Library and explicitly insert it as the AST node instead of simply using Java `null` value. The problem with this, however, is that attribute evaluator does not have direct access to the OCL Library in the repository (access is only possible via a model element, e.g. Classifier, which is an instance of a concrete implementation of one of the `ICommon` interfaces). We could, in principle, retrieve the top-level package of the whole model, and access `Ocl Library` via model element `Package`.

But a much neater way is using the `TypeEvaluator` (used for type inference) which is associated with our instance of `LAttrAstGenerator`. `TypeEvaluator` has access to the abstract interface of the `OclModel`, which is implemented by a concrete instance of `OclLibrary`. Hence we retrieve the `OclVoid` type by implementing and calling `getOclVoid()` method, provided by `TypeEvaluator`. The latter calls a similar already-defined method to retrieve the actual type from the OCL Library.

**Extending Access of Association Ends and Classes**

The problem with the representation of the integrated UML/OCL metamodel in the original Dresden OCLParser, is that it does not implement some methods for access of **association classes** and **association ends**. During attribute evaluation and for representing the actual tree of the integrated metamodel in OCL Evaluation Framework, we are interested in showing which model elements have access to association ends and classes. **Figure 3.5** illustrates an extract of the integrated UML/OCL metamodel displayed by original OCLParser for our Car World Example from **Figure 2.2**

As can be seen, the representation provides an incomplete picture of relationships between model elements, because e.g. one can navigate from class `Person` to a set of instances of `Car`
via the ownedCar association end of association Ownership. In the integrated metamodel association ends and association classes are not visible from classifiers, which participate in associations, even though in OCL these association ends and classes can be navigated from classifier context (e.g. context Person inv: self.ownedCar... navigates association end ownedCar from classifier Person). The tree is built using a utility class, which calls corresponding methods, implemented by the ICommon interface. E.g. to display all operations contained in the given classifier (Person in Figure 3.5), it simply calls classifierInstance.allOperations() method, which is implemented by ClassifierImpl.

Hence the problem is that ClassifierImpl provides implementations of allOperations() and allAttributes(), but does not implement allAssociationEnds() and allAssociationClasses(). Implementation of these methods is also non-trivial compared to, say, allOperations() which consists of a loop which calls getOperation() and does the necessary processing at each step. The general algorithm can be exemplified for allAssociationClasses() as follows:

1. obtain the ModelFacade instance, which is based on the Facade design pattern, and automatically “links up” the call with the currently active implementation of the repository.
2. get the list of all existing association ends in the repository; this method is already implemented by the MOF repository. The next part of the algorithm is based on constructing a set of association classes, which relate to associations, that our classifier participates in.
3. iterate through association ends, and for each association ends, we retrieve its association (assocEnd.getAssociation()).
4. accumulate association classes in a separate list, and check whether each association we retrieved in the previous step, can be cast to an association class. Since user-defined
MOF representations are not supported by our repository yet, we implement this for UML models:

```java
Association a = assocEnd.getAssociation();
if (a instanceof UmlAssociationClass && !assocClasses.contains(a))
    assocClasses.add(a);
```

We also ensure that our accumulated list does not already contain the association class. This is needed for recursive associations (i.e. associations between the class and itself): in this case, there are two association class instances - one for navigating the association from left to right and one for right-to-left navigation. Hence, for recursive associations, the if-statement above would be “hit” twice. Thus we need to ensure we add the association class into the representation only once.

5. Finally, call this operation `allAssociationClasses()` recursively on each parent of the given classifier. This is due to inheritance, i.e. our classifier can inherit associations and hence association classes of parents via `generalization` in UML.

The algorithm is similar for `allAssociationEnds()`, and will be omitted.

The newly implemented methods give us a straightforward interface for accessing association ends and classes for a given classifier and hence displaying a complete integrated UML/OCL metamodel, which exhibits relationships between model elements in exactly the same structure as their navigation in OCL syntax.

**Implementing Type Conformance Operations**

In addition to `allInstances()`, other operations with “special” syntax provided in OCL are the type conformance operations `oclIsKindOf()` and `oclIsTypeOf()`. They are special in that the single argument that they take is not an instance, but a classifier, e.g. `self.oclIsKindOf(OclAny)` (much like the source of the `allInstances()` call in `OclAny.allInstances()`).

The contextual evaluation functionality which needs to be implemented, is similar to that of `allInstances()` which has already been discussed above. For this reason, we will omit implementation details here.

It is important to note, that these operations are not supported in the Dresden OCLParser, but have been implemented correctly on both syntactic parsing and contextual evaluation levels in our OCL Evaluation Framework.

This completes the design of our syntactic evaluation subsystem, which is part of OCL Framework. In the next chapter we describe the design and of our semantic evaluation subsystem, which has been implemented from scratch and is entirely independent of Dresden OCL Toolkit.
Chapter 4

Specification and Design: Semantic Evaluation

In this chapter we present our semantic evaluation subsystem, developed as part of OCL Evaluation Framework. Unlike syntactic evaluation, which was developed by extending Dresden OCLParser, this subsystem was developed from scratch and integrated together with syntactic evaluation to work under one front-end.

Firstly, we describe the exact functionality of our system. This is followed by an overall view of design, which is then described in greater detail in subsequent sections.

4.1 Specification

Semantic evaluation is implemented on top of syntactic evaluation, i.e. it requires the latter to be performed beforehand on OCL constraints, entered by the user. During contextual evaluation, constraints are accumulated in the required form for semantic evaluation (which we will refer to as semantic form) at the same time as the AST is built, i.e. in total, only one traversal of the CST (entered OCL constraints) is performed. After semantic evaluation completes, the Knowledge Base is displayed as a tree (exemplified in Figure 4.1).

Semantic evaluator processes OCL constraints to infer a set of true facts about the system, which constitute the Knowledge Base (KB). Consequently, each constraint is evaluated against constraints which are already in the KB (i.e. the base case is evaluating a constraint against an empty KB):

- if the given constraint contradicts the KB, then it is established as a false fact and is not added to the KB (it would marked in red color in the tree).
- if it does not contradict the KB, it is define to be a true fact. The evaluator then checks whether this constraint introduces any new semantics, i.e. whether it is not already implied by the KB facts.
- if the constraint is already implied by the KB, then it is established as an excessive fact (and would highlighted in yellow in the tree), in which case it is not added to the KB. Otherwise, it is highlighted in green which denotes a non-contradictory constraint, that introduces new semantics to the KB. An excessive constraint does not, however,
Figure 4.1: Screenshot of Knowledge Base in OCL Evaluation Framework
violate the KB. We choose not to add it since it introduces no extra meaning into the KB.

We have just described our approach to evaluating satisfiability of constraints, which is the main function of our semantic evaluation subsystem.

Constraints are added to the KB in an order, which is in general different from the order in which the user specified them. The reason for this is due to the fact that we evaluate each to-be-added constraint against constraints which are already in the KB. Thus, the result of semantic evaluation crucially depends on which constraints our to-be-added constraint is evaluated against. We discuss and motivate the order for addition to the KB later in this chapter.

The scope of our semantic evaluation can be specified as follows:

- *all OCL constraint types* are supported: invariants, pre- and post-conditions, operation bodies, initial and derived values, attribute and operation definitions
- *all imported UML types* are supported; all types which are imported via UML class models by the user, are supported in semantic evaluation
- *built-in OCL types* Integer, String, Boolean are supported. All collection types (sub-types of abstract type Collection in OCL) are not supported. Any OCL expression, which involves collection types therefore is not supported either
- *built-in OCL operations on types* Integer, String, Boolean are supported (we have provided all supported operations in Table 2.1 in the Background chapter). Since collection types are not supported, operations on them are not supported either
- operations on OclAny are not supported. Since OclAny does not in general evaluate to a semantically meaningful type, semantic evaluation procedures are unclear. Hence, its operations are not supported (yet) in the semantic evaluation subsystem. We leave this as future work

This completes our description of functionality of semantic evaluation subsystem in OCL Evaluation Framework. The next section provides the overall structure of the design, which is expanded in the subsequent sections.

### 4.2 Overall View of Design

*Figure 4.2* illustrates the structure of the subsystem, which comprises a set of Java classes, a Knowledge Base Evaluator, implemented in SICStus Prolog, and a Wolfram Mathematica kernel, which is used to reduce sets of numerical relations and functions. We can describe the main components as follows (referring to *Figure 4.2*):

- **GUI and LAttrAstGenerator** are parts of the syntactic evaluation subsystem, which we discussed in the previous chapter. The graphical user-interface is the front-end to the whole system, OCL Evaluation Framework, while LAttrAstGenerator is the attribute evaluator class, which implements contextual evaluation. As we mentioned earlier, it essentially constructs the AST in one traversal (using abstract syntax of OCL Metamodel). During the same traversal, it calls corresponding methods on the associated instance of **ConstraintPacker** for converting constraints into the semantic form and packing them, so that they can later be injected into the Knowledge Base.
Figure 4.2: Design of the Semantic Evaluation Subsystem

- **Evaluator** class provides the main interface to semantic evaluation subsystem, which is used by the front-end to obtain an instance of **ConstraintPacker**. The latter is then used with the attribute evaluator. The Evaluator delegates the front-end request to generate the knowledge base tree to the **ConstraintInjector** instance. Both, **ConstraintPacker** and **ConstraintInjector**, are singletons.

- KB is generated by a sequence of calls from **ConstraintInjector** to the **main.pl** component of the KB evaluator, implemented in SICStus Prolog. This is done via the **Jasper** (Java→Prolog) interface, provided by SICStus. **main.pl** calls the goal-directed theorem prover in **derive.pl** to infer whether the constraint (already in semantic form) can be added to the KB, and whether it is excessive. **print.pl** is the printing module, which is used to convert the internal Prolog representation of the KB to a human-readable format. The internal output is thus passed to **print.pl** which converts it into a form, in which the output ultimately appears in OCL Evaluation Framework front-end (as part of “Constraint Injection Log”, as shown in Figure 4.1).

- as mentioned earlier, our goal-directed theorem prover is extended (we presented the theory behind propositional goal-directed theorem provers in Section 2.3), which refers to the fact that as well as handling logical connectives (and, or, implies...) it can also process **numerical functions** (e.g. 2 + 5, 3 − (5/6) * 3) and **relations** (e.g. 5 > 6, 3 ≤ 5/6 * 3). Numerical functions and relations are not handled in Prolog, and therefore our prover (**derive.pl**) delegates this processing to **mathcall.pl**. The latter converts internal prolog representation of KB facts into a suitable form, which is allowed by Wolfram Mathematica, and calls **MathCall** class via Jasper (Prolog→Java) interface. MathCall is a proxy class, which simply passes on the string command received from **mathcall.pl** to the **Mathematica 6 kernel**, via the **JLink** interface (Java→Mathematica), provided by Wolfram Research. **mathcall.pl** receives the response, and with minimal processing (to “make sense” of what is returned by Mathematica) passes it back to the goal-directed theorem prover in **derive.pl**.

- When injection is complete, **ConstraintInjector** retrieves the output from **main.pl** and
updates the KB tree representation in the front-end of OCL Evaluation Framework.
The iteration, which we have just described, is performed for each constraint until the
KB tree for OCL constraints entered by the user, is built-up completely (as shown in
Figure 4.1).

In the following sections, we describe the implementation of each of the components from
Figure 4.2 in greater detail.

4.3 Packing Constraints

In addition to the syntactic evaluation subsystem, we have developed a set of classes to
manage semantic evaluation from the Java side. These classes are part of the package
peterlev.semantics.analysis in source code. Evaluator class provides the interface to the
implemented facility, which comprises ConstraintPacker and ConstraintInjector. In this
section, we describe the functionality and design of ConstraintPacker, which is used to
convert constraints to semantic form and collection them for injection.

The main challenge that we face is that our Knowledge Base Evaluator, written in
Prolog, accepts constraints in a different form, i.e. in the form of prolog predicates (e.g.
equals(a,b) could represent a=b in Prolog), which can then be easily manipulated within
Prolog. For this reason, passing constraints in the same form as they are specified by the
user (in OCL 2.0 syntax) is insufficient. For an optimal solution, we are also constrained by
the fact that the AST is constructed in one traversal (and attribute evaluation is performed
simultaneously) and it would be tedious and inefficient to convert constraints to semantic
form and pack them for injection using a separate, and hence, additional traversal. We
propose a stacking method for constructing semantic form for constraints within the same
single traversal of the CST, as the one used for constructing the AST by our attribute
evaluator LAttrAstGenerator (discussed in the previous chapter).

We illustrate this method with an example, omitting the details of how semantic form
is constructed for the time being. Consider the following post condition in OCL, defined on
operation getTopSpeed() (as modeled by our Car World example in Figure 2.2):

context Car::getTopSpeed() : Integer
post: if A

then B
else C endif = result

Let us abstract away from what A, B and C really are. In general they can be any OCL
expressions, except that we can infer that A must evaluate to Boolean (since it is a condi-
tion), while B and C must be of type Integer (since they are equated to result, which has
the operation return type), given that the expression is specified correctly. Note that when
executing semantic evaluation, we must ensure that the constraints themselves are correctly
specified (otherwise syntactic and contextual errors can propagate into our KB evaluator),
and this forms the main reason why we oblige the user to perform syntactic evaluation before
semantic evaluation in OCL Evaluation Framework. This also “relieves” our semantic evalu-
ation subsystem from checking types, operation consistencies etc because the pre-conditions
to semantic evaluation now assert that the OCL syntax must be correct at time of semantic
evaluation.

Back to our OCL example, let us consider the concrete syntax tree (CST) associated
with the OCL expression (hence we omit post and context nodes from the tree), which is
illustrated in Figure 4.3.
We can compare this with Figure 3.4, which showed a similar tree diagram to illustrate how our attribute evaluator performs contextual evaluation via `computeAstFor...` methods, which are executed on exit of each AST node, that in turn is generated for each CST node. The reason for this similarity is due to the fact that semantic form for constraints is computed at the same time as the context is evaluated. Our “computeAst...” methods were modified with entry calls to ConstraintPacker, which then computes the semantic form. All semantic forms are Strings in Java. Let us go through the computation of semantic form, illustrated in Figure 4.3:

1. recall that our traversal algorithm uses left-to-right depth-first order, which means node for expression A is processed first. The corresponding `computeAstFor...` method for A, which computes synthesized attribute, when the AST node for A is exited, can be described by the following pseudocode (we abstract from tedious implementation details):

   ```java
   computeAstForA(...) {
     checkingTypesForA();
     calcSAforFutureNodes();
     // entry hook to ConstraintPacker
     sPack.addA(nameA, typeA, ...);
   }
   ```

   Once SA are computed, we call an associated instance of ConstraintPacker to create a semantic form for A via `sPack.addA()` (we pass expression-dependent parameters, e.g. for attribute this would be attribute name and attribute type) and add it onto the stack, which will be used for collecting all constraints. We refer to this stack as `semantic stack` for the sake of compactness. This is illustrated by $A_s$ in stack state (1) in Figure 4.3.
2. The next node to be traversed is B, which results in $B_s$ being added onto the semantic stack in a similar fashion.

3. Next, C is processed. On exit of node C, the stack already contains (top-to-bottom order) $C_s, B_s, A_s$.

4. The next node which is exited by our tree walker corresponds to if-A-then-B-else-C expression from our OCL example. The relevant parts of the method for computing SA for if-then-else expressions is given as follows:

   ```java
   public IfExp computeAstFor_AIfExpCs(Heritage hrtg, ...) {
   ... // SA are computed
   Classifier ctx = hrtg.getContextualClassifier();
   ModelElement consElem = hrtg.getConstrainedElement();
   // entry hook to semantic evaluation here
   sPack.addIfExpr(ctx, consElem, astElseBranch != null);
   // return AST node
   }
   ```

After SA routines, we call ConstraintPacker instance to create a semantic form for an if-then-else expression, given its context (Car in our example) and constrained element (operation `getTopSpeed()`). Note that we also indicate whether an else case has been specified (in OCL, else is optional, i.e. one can specify only if-A-then-B) by checking whether the else branch (instance of AST node for else), which is passed to us from node C is not null (if it is, AST node does not exist, i.e. there is no else). `addIfExpr(...)` starts by popping the last three members off the stack (if else was specified, 2 otherwise). We are guaranteed that these must be semantic forms for C, B and A, since no other violating node could have had access to the tree before this node (if-then-else node). Conversion of if-then-else to semantic form is not straightforward due to the fact that there is no direct representation for this in logic (as opposed to, say, implies in OCL, which is converted to $\rightarrow$ in logic). But we can use the following semantically equivalent form: if-A-then-B-else-C $\Rightarrow$ $A \rightarrow B \land \neg A \rightarrow C$ (Translation 1). This is sufficient only for cases where B and C are Boolean. In all other cases, further modifications are required. At this stage, let us assume this discrepancy is already dealt with (“leap of faith”) and that we have flagged to the attribute evaluator that an if-then-else expression has just been processed.

5. We then traverse the node on the same depth level, which is result. Similar to cases above, it gets converted via a call to `sPack.addVar(varName, varType)` in the attribute evaluator, where `varName` is “result” in this case and `varType` is the type of the variable, inferred by calling an instance of `TypeEvaluator (typeEval.obtainType(varAST))` to obtain the type of the corresponding AST node (which is `VariableExp` for variables, as in our case). The semantic form for result is `varlet(result, [type])`, where type is a common supertype of A and B (e.g for A = 5 and B = 4.5 the common supertype is `Real`). The common super type is inferred by TypeEvaluator as described above. `result` is not an operation, i.e. it does not have operands and therefore does not pop elements off the stack. Hence, `varlet(result, [type])` is pushed onto the stack and we obtain stack state (3) in Figure 4.3.

6. Finally, we exit the node for “=”, which is a binary relation. Therefore two members are popped off the stack, and they are guaranteed to specify operands to “=” in reverse order (due to FIFO nature). Similar to previous cases, attribute evaluator’s
corresponding method for processing numeric relations (computeAstFor...) contains an entry hook to the corresponding method \texttt{addArithmeticExpr(String opName)} in \textit{ConstraintPacker}, which constructs the semantic form. In general, two expressions B and A would be popped off the semantic stack, and semantic form eq(A, B) (for \( A = B \)) constructed and pushed onto the stack. However, recall that in point 4 above we mentioned the “leap of faith” for if-then-else expression: initially, our method checks whether an if-then-else expression has just been processed, which is true in this case. This results in a different routine for constructing semantic form. Essentially, we want to transform \( A \rightarrow B \land \neg A \rightarrow C \) into \( A \rightarrow \text{result} = B \land \neg A \rightarrow \text{result} = C \), which validates our expression, since consequents of both implications now evaluate to type Boolean. This results in obtaining stack state (4), which completes the packing of our example OCL expression.

7. When we process the node for “post” (from our OCL example above), we are guaranteed to have the associated OCL expression residing on top of the stack. We pop it and use it to construct the semantic form for the whole constraint (done in \textit{ConstraintPacker}), which is added to the list of all constraints, that have been accumulated.

All other expressions and constraints are converted into semantic form in a similar fashion. The advantage of this method is that it introduces no time overheads to syntactic evaluation, because constraints are gathered in the same traversal of the AST as the one, which contextual analysis is performed in. Note that for our if-then-else example stack state 3 of \textit{Figure 4.3} is not entirely accurate. We have simplified the process for if-then-else expressions to emphasize how previously added elements are popped and manipulated at each node of the AST, and the new form is pushed onto the stack. For if-then-else expressions, node “if-then-else” does not actually push the conjunction of implications onto the stack (as illustrated in stack state 4). Instead, it leaves C, B and A intact on the stack, and pushes an “IFEXP” (or “IFELSEEXP” when an “else” case is included) string token onto the stack, which indicates that the if-then-else expression should be processed by its parent. The reason for that is, as mentioned, B and C in \textit{Translation 1} specified above, are not in general Boolean, therefore to make Translation 1 valid we need to perform the following transformation in the general case (if-then-else is a right operand): \( \text{leftOp} - \text{op} - (A \rightarrow B \land \neg A \rightarrow C) \Rightarrow A \rightarrow (\text{leftOp} - \text{op} - B) \land \neg A \rightarrow (\text{leftOp} - \text{op} - C) \) (Similar when if-then-else is a left operand). We have exemplified this in \textit{Figure 4.3} for “=". The real contents of stack state (3) is \{\textit{result*}, IFEXPELSE, C, B, A\}. Hence, after popping \textit{result*}, we know that the right operand is an if-then-else expression, and hence we have enough knowledge to produce stack state (4).

4.3.1 Converting Constraints to Semantic Form

We have described how packing is performed and have up until this stage ignored the exact procedure for construction and the structure of semantic form. The structure of semantic form has been designed in such a way, that its corresponding String representation in Java, can be passed to KB Evaluator (implemented in Prolog) directly, and can therefore be decomposed into a Prolog fact, e.g. equals(numexp(5), numexp(5)) in Prolog has the semantics of representing the fact that 5 = 5 (note: it does not indicate whether 5 = 5 is true or false). For each expression type in OCL, ConstraintPacker contains corresponding methods for computing semantic form:

- \textit{Boolean}; a typical entry hook in the corresponding \texttt{computeAstFor...} method of our attribute evaluator calls \texttt{addBoolean(boolean b)}, which pushes string “boolean(b)” onto the stack.

69
### Table 4.1: OCL Numerical Expressions and their Semantic Forms

<table>
<thead>
<tr>
<th>Functions</th>
<th>Semantic Form</th>
<th>Relations</th>
<th>Semantic Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A + B$</td>
<td>plus(A*, B*)</td>
<td>$A &gt; B$</td>
<td>gt(A*, B*)</td>
</tr>
<tr>
<td>$A - B$</td>
<td>minus(A*, B*)</td>
<td>$A &lt; B$</td>
<td>lt(A*, B*)</td>
</tr>
<tr>
<td>$-A$</td>
<td>negat(A*, B*)</td>
<td>$A = B$</td>
<td>eq(A*, B*)</td>
</tr>
<tr>
<td>$A * B$</td>
<td>mult(A*, B*)</td>
<td>$A &lt;&gt; B$</td>
<td>neq(A*, B*)</td>
</tr>
<tr>
<td>$A / B$</td>
<td>divide(A*, B*)</td>
<td>$A &gt;= B$</td>
<td>ge(A*, B*)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$A =&lt; B$</td>
<td>le(A*, B*)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operations</th>
<th>Semantic Form</th>
<th>Operations</th>
<th>Semantic Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.abs()</td>
<td>absolute(A)</td>
<td>A.div(B)</td>
<td>quot(A, B)</td>
</tr>
<tr>
<td>A.mod(B)</td>
<td>remain(A, B)</td>
<td>A.floor()</td>
<td>floor(A)</td>
</tr>
<tr>
<td>A.round()</td>
<td>round(A)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Integer, Real**: follow the same procedure, as they are converted into “integer(n)” and “real(n)” for original value n and pushed onto the semantic stack, in exactly the same fashion as we have described up until this point.

- **String**: In OCL, strings are specified as e.g. 'peter'. The corresponding semantic form is 'string('peter')'.

- **Logical Connectives**: `and`, `or`, `xor` and `implies` are binary operators, which are converted using `addLogicalExpr(String opName)` in `ConstraintPacker`. Since they are binary, the method pops two members off the stack, which represent operands of the given connective in reverse order (e.g. B, A for `and(A,B)`). Prolog facts (A and B), (A or B) etc are constructed and pushed onto the stack. They could be pushed in prefix notation as and(A,B) and or(A, B), but we adopt the former convention for greater readability. Negation `not` is a unary operator however, hence we use exactly the same procedure but pop only one member A off the stack and construct the corresponding semantic form as (`neg A`).

- **Numerical Functions and Relations**: The processing is performed exactly the same way as for logical connectives. Binary operators `+`, `−`, `∗`, `/`, `>`, `<`, `>=`, `=<` are constructed by popping B, A and pushing `plus(A,B)`, `minus(A,B)`, `mult(A,B)`, `gt(A, B)` etc depending on the given operator. The full list is given in Table 4.1 (* denotes semantic form). The only difference from logical connectives is that infix forms, e.g. (A plus B) are not supported. This is done to distinguish between logical and numerical operations visually: semantic forms for logical connectives use infix notation, while prefix notation is used for numerical functions and relations.

- **Numerical Operations**: These are formally defined as operations in OCL, and therefore can only be called in postfix syntax. Semantic forms for numerical operations are illustrated in Table 4.1. While, numerical relations and functions are processed by `addArithmeticExpr(String opName)` in `ConstraintPacker`, operations are processed using `addArithmeticOpExpr()` simply for conceptual distinction, although their implementations are almost identical. These two operations, however, are called from different attribute evaluation methods in LAttrAstGenerator. `addArithmeticExpr()` is called from `computeAstFor...` method for binary relations (and unary) relations, while `addArithmeticOpExpr()` is called from `computeAstFor...` method for computing paths (e.g. `-5.abs()` is evaluated as a path from instance of `Integer` to operation `abs()`, defined within class `Integer`), which is used for all operations. Hence we explicitly try and match the given operation name to one of the operation names from Table 4.1.
and call addArithmeticOpExpr() hook if succeeded. Otherwise, we use routines for processing any generic operation (e.g. defined within UML class model).

• **Operations.** The semantic form varies on whether *operation signature* or *operation call is specified*. The general form for operation signatures is oper(name, [context],[argumentTypes],[returnType]). E.g. for operation `carworld::Car::driveTo(x:Real, y:Real) : Boolean` the corresponding semantic form is "oper(driveTo, [carworld, car],[[real],[real]],[boolean])". For operation calls, the same structure is maintained, except instantiated operands are specified instead of argument types. An operation call `driveTo(5, 4.5)` would be exactly the same, except `[[real],[real]]` is replaced by `[numexp(5), numexp(4.5)]`.

• **Attributes** are expressed with similar semantic form “attr(name, [context], type)”. E.g. attribute `carworld::Car::age : Integer` corresponds to “attr(age, [carworld, car],[integer])” in semantic form. Note how paths to the attribute/operation are converted into semantic contexts: `(carworld::Car) ⇒ [carworld, car]`. Semantic form always uses comma separated lowercase names ([A, .., B] represents Prolog list directly and hence can easily be manipulated within KB Evaluator).

• **Built-in OCL Types** are converted without any reference to context, e.g. “[real]”, “[integer]”, “[boolean]” and “[string]” denote semantic forms for OCL Library types Real, Integer, Boolean and String accordingly. (Recall, that we do not implement other, i.e. non-collection, types in semantic evaluation subsystem).

• **UML Model Types** must be specified with their associated context to avoid possible naming ambiguities. We have already exemplified contexts for attributes and operations. Classifiers, e.g. Car in package carworld, cannot appear as final types of any OCL expression which involves only non-collection types. The reason for that is that any reference to one classifier from another (e.g. Car can reference Person) is defined as an association in the UML class model, which is represented as a collection (Set) in OCL.

• **Constraints.** The general semantic form for OCL constraints is “constr(name, [type, context], expressions*)”. Name is optional, and if unspecified is set to 'unnamed'. Type could be one of `inv` (invariants), `pre`, `post`, `def` (attr/oper definitions), `init` (initial values), `derive` (derived value expressions), `body` (operation bodies). context specifies the constraint element, and expression* is the associated OCL expression already in semantic form. E.g.:

```plaintext
context carworld::Car : age : Integer
    init : 6
```

Corresponds to
"constr('<unnamed>',[init,attr(age, [carworld, car],[integer])],numexp(6))" in semantic form. Context is given by the semantic form for the corresponding constrained element (in our case, we provide semantic form for attribute `age`). In reality, the semantic form we have provided for the example is incomplete.

### 4.3.2 Constructing Boolean Expressions from Constraints

As we have just mentioned, the semantic form for the OCL constraint above is incomplete. The reason behind this is that expressions in semantic form must always evaluate to type
Boolean, because semantic forms are created to represent facts (which are either true or false) in our Knowledge Base (in Prolog). We briefly discuss how each type of constraint is converted into an expression which evaluates to type Boolean (implementation details are omitted, since they involve trivial string manipulations):

- **Invariants.** No additional manipulation is required, since the associated OCL expression by definition evaluates to type Boolean.

- **Pre-conditions.** Do not require conversion to type Boolean, since they involve Boolean expressions already. However, we can improve on cases, which specify pre-conditions on the same operation. The problem occurs because of potentially different arguments naming, e.g.:

  ```
  context Car::driveTo(x:Real, y:Real) : boolean
  pre: x > 0 and y > 0
  ...
  context Car::driveTo(a:Real, b:Real) : boolean
  pre: a < 0 and b > 0
  ```

  This example specifies two contradictory pre-conditions for the same operation. However, if they are converted to semantic form as-is, our KB evaluator will not be able to infer that in fact a=x refers to the first argument, and b=y to the second argument of the operation. Hence we perform a further transformation to replace all argument references by opArg1 and opArg2 for both pre-conditions. This is achieved by trivial string manipulation functions, as we replace "varlet(x,[real])" and "varlet(a,[real])" with "varlet(opArg1,[real])" in the string, representing semantic form for the constraint.

- **Post-conditions.** No conversion is required. However, we can improve cases, in which the result variable is used (e.g. `result = self.topSpeed`). Recall that it denotes the return value of the operation and hence can be replaced by the operation signature itself, i.e. `result = self.topSpeed ⇒ oper(getTopSpeed, [carworld, car], [], [real]) = attr(topSpeed, [carworld, car], [real])`. In this conversion, we have employed semantic forms for operation signature of `getTopSpeed()` and attribute `topSpeed` (in accordance with rules specified above). Also, argument name replacing transformations are performed the same way as for pre-conditions.

- **Operation Bodies.** Since body: `expr` in OCL is equivalent to post: `result = expr`, we use the procedure outlined for the result variable above to convert operation body into an expression of type Boolean.

- **Attribute/Operation Definitions.** These are assertions, which are practically expressions which should always evaluate to true in the system, e.g. `def kmsCovered : Integer = 100`. The semantic form is therefore trivial: “attr(kmsCovered, ...) = numexp(100)”. 

- **Initial and Derived Values.** These expressions specify values of attributes/association ends, which in general are not of type Boolean. Semantically, they define the value of the attribute/association end. This gives us the required conversion rule: e.g. `context attrName:type init: value ⇒ “attr(attrName, ..., type) = value*”`, which now defines a fact (either true or false). `value*` denotes the value of the attribute given in semantic form.
Using these conversion rules, we represent OCL constraints as facts in semantic form, which evaluate to either true or false. This is the final form required by our KB Evaluator (and the theorem prover in particular). We have thus described all procedures involved in collecting (packing) constraints in semantic form.

4.4 Injecting Constraints into the Knowledge Base

By virtue of constraint packing, which we have just described, we now have access to a list of all constraints in semantic form, in the same order as the one in which they have been entered by the user. However, constraints are inserted into the Knowledge Base in a different order: since we evaluate each constraint against all previously added constraints, the satisfiability outcome crucially depends on which constraints are already in the KB at the time of evaluation of the given constraint.

We have thus specified the order for insertion of constraints into the knowledge base, which we will refer to as semantic order.

4.4.1 Semantic Order

In general, user can specify OCL constraints in any order. We are interested in constructing an order, based on semantic “strength” of constraints. We propose and motivate this order as follows (\{a, b, c\} is used to indicate that expressions a, b and c can appear in any order, e.g. in the order they have been specified by the user):

1. definitions must be inserted into the knowledge base first. They cannot potentially contradict any KB facts, since they are merely assertions - definitions of helper operations and attributes, that might be used by other constraints. On the other hand, if they had not been inserted beforehand, and a constraint is added, which involves these helper operations/attributes, then our KB evaluator will not employ the definitions of these helpers to reason on the constraint being added (i.e. it will assume that nothing has been said about the helpers, which the constraint employs). This can result in contradictory constraints being inserted.

2. \{invariants, initialvalues, derivedvalues\} should be inserted afterwards. They can follow the same order, in which they have been specified by the user, because they are of equivalent semantic “strength” (they should always hold): invariants specify statements that should hold in any state of the system, initial values specify expressions of attributes that should always hold at initialisation of the system, and derived values specify expressions of attributes that should hold in any state (similar to invariants). Hence we could have situations where a user specifies an invariant, which fixes a value of an attribute (e.g. age = 5), and then a contradictory initial value expression for that attribute (e.g. context ..age.. init: 6). On the other hand, he could specify the initial value expression first, and it would then be contradicted by the invariant, specified later. This example exhibits the fact that there is no one “right” ordering between invariants and initial values, so we allow any order. Similarly, derived values for an attribute can contradict previously specified initial values and vice versa.

3. pre-conditions are of weaker “semantic” strength since if they are contradicted (worst case), the operation, on which they are specified, never gets executed (on the other hand, if we specify contradictory invariants, the whole specification of the system is infeasible). Contradictory pre-conditions should not be added to the Knowledge Base: if they are false, one could say the operation never gets executed (suppose it checks...
for pre-conditions in code, and fails if they are not satisfied) or, alternatively, the effects of the operation call are *unpredictable* (in logic: pre-conditions derive falsity, and falsity implies any post-condition (valid or invalid) for the operation, hence our KB itself can become contradictory). Whichever semantics one sides with, there is no sense in inserting the pre-conditions into the knowledge base. Furthermore, once we know pre-conditions are contradictory, post-conditions for the same operation should not be considered by the KB Evaluator. For the *call-not-executed* semantics, the set of post-conditions is simply \( \{ \top \} \), i.e. user-specified post-conditions are ignored since the operation makes no changes to the system, as it is not executed. Appending \( \top \) to the KB introduces no extra semantics and hence can be avoided. For the *call-unpredictable* semantics, actual user-specified post-conditions are irrelevant, since any post-condition now holds true (as pre-conditions derive \( \bot \)). Hence, we avoid the risk of violating the correctness of KB by ignoring post-conditions altogether. We have also motivated the need to insert pre-conditions before post- (and therefore before operation bodies), since if the former are contradictory, the latter should not even be considered. Any other order can result in contradictory KB.

4. \( \{ \text{postconditions, operationbodies} \} \) are therefore inserted last and can follow any order, because they have almost identical semantics (both specify return value of operation), except that post-conditions can also be used to assert other facts about the system. Furthermore, if pre-conditions for the operation have already been determined as contradictory, post-conditions and operation body specifications for the same operation should be ignored.

We can summarise the semantic order as 
\[
def, \{ \text{inv, init, derive} \}, \forall x (\text{pre}(x) \rightarrow \{ \text{post}(x), \text{body}(x) \})
\]

We re-order the original list, which follows order of user specification, to satisfy the semantic order stated above (the implementation involves trivial list manipulations in Java).

This list is then passed from ConstraintPacker to ConstraintInjector (recall Figure 4.2), which uses its main routine, orderedInject() to inject the constraints from this list into the Knowledge Base one at a time.

### 4.4.2 Injection Method

ConstraintPacker class contains different types of methods. The main methods are orderedInject() for performing injection of constraints in semantic form into the Knowledge Base, and generateKB() for generating the KB tree, which is then displayed in the front-end of OCL Evaluation Framework. The tree is generated from a list of ConstraintState instances. Each instance comprises constraint name, type, context and body, as well as the injection log, which is the output from our KB evaluator. It is displayed to the user, so that one could analyse the reasoning of our goal-directed theorem prover as it establishes whether the constraint was contradictory (and if not, whether it was excessive).

Most of the functionality comes “alive” in the orderedInject() method, which is presented below via one generic iteration. We abstract away from routine and trivial implementation details in Java.

1. **Obtaining an instance of Prolog Runtime System.** We implement accessors using singleton pattern to ensure that only one instance of Prolog is running. Prolog runtime is obtained via SICStus Jasper interface (as illustrated in Figure 4.3), and represents the same instance of Prolog as one would obtain by running the Prolog window, except that our Prolog runtime is called from within Java. The prolog runtime was created
from the set of *.pl files, which constitute the KB Evaluator, and compiled into a single file “startup.po”, which is run from Java via

1. Prolog plg = Jasper.newProlog(null, null, 
   "’/kb_eval/startup.po’");
2. Term mathObj = plg.newObjectTerm(getMath());
3. vars.put("MathObj", mathObj);
4. if(plg.queryCutFail("mathsetup(MathObj).", vars))
   // print success msg to console

2. Obtaining MathCall proxy. MathCall is instantiated using singleton pattern, similar to Prolog, and represents a simple proxy class, which provides access to Mathematica kernel. The kernel will be required by KB evaluator, and could be instantiated by the evaluator directly. However, this would require a separate Jasper instance. Hence ConstraintPacker creates an instance of MathCall and passes its reference to the KB evaluator in Prolog. The code illustrates this: vars is a HashMap used to store variables and their associated bindings, which will be sent to/received from Prolog via the Jasper interface (e.g. X = 5 binds variable X to 5 in Prolog; it would be received in Java as HashMap entry ((String) “X”, (int) 5)). We obtain the MathCall instance via a call to getMath(), and create a Prolog object term for it on line 2. We then insert the object reference as a binding to variable MathObj into the HashMap. On line 4, we execute a predicate mathsetup/1 in our KB Evaluator, providing it with the object reference. Note that we use queryCutFail, which simply executes a query and does not backtrack to find any solutions (in our case, we do not require any bindings). mathsetup/1 is defined in mathcall.pl (see Figure 4.3), and asserts mathObj(reference) as a fact in the Prolog system, so that it could always be accessed by computational methods in mathcall.pl in order to forward calculations to Mathematica kernel. MathCall implements one main method evalExpr(String command), which takes a command as a String (already in Mathematica form) and returns a String representation of Mathematica output. The kernel is called via the Wolfram JLink interface, which provides a simple high-level interface from Java to Mathematica. The computation is performed by one call: String strResult \= ml.evaluateToOutputForm(command, 0), once the kernel KernelLink ml has been set up.

3. Extensions. We will discuss these later. Extensions are hooks to other methods, to perform some “maintenance” work for specific cases. For each iteration, they are executed before we start processing the injection of a constraint.

4. Configuring prolog environment. Our KB evaluator allows certain flexibility, which can be controlled by predicate setEnv/3 (part of main.pl), which can be used to change some of the environment settings: whether to print injection log to the Java console or flush it to a buffer (array of characters), whether to actually insert satisfactory constraints into the KB (one could simply use testing mode, when constraints are only evaluated but not injected), and whether to print a detailed form of the log (which includes the goal-directed derivation details). We adjust the environment to insert constraints as facts into the KB, produce detailed injection logs and print the logs into a buffer (so we can retrieve it once injection is complete, to obtain the injection log).

5. Injecting constraint. Constraint is already in the semantic form by virtue of constraint packing.

1. Prolog p = this.getPrologRuntime(); // singleton access
2  Query query = p.openPrologQuery(
3         "inject("+constr+", S, Ex.").", vars);
4  try {
5      if(query.nextSolution()) { // has solutions?
6          Term t1 = (Term) vars.get("S");
7          Term t2 = (Term) vars.get("Ex");
8
9          if(t1 != null && t2 != null) {
10             out1 = t1.getBoolean();
11         } // else error
12
13         ...  

Hence, the only task left is to call inject/3 in main.pl (line 2). This time we provide
input bindings for “S” and “Ex” in the vars HashMap, since we need to receive answers
(output bindings) from KB evaluator (lines 6 and 7). “S” (boolean) indicates whether
the constraint has satisfied the KB facts, and “Ex” (boolean) specifies whether the
constraint was excessive (only meaningful if S=true). These two answers give Con-
straintInjector enough information to infer the exact status of the constraint in the
Knowledge Base. All answers are stored in ConstraintInfo object, which is created for
each constraint. Note that the prolog call is openPrologQuery (line 2), which allows
us to backtrack to different solutions (even though our Prolog implementation guar-
antees to return only one set of variable bindings for inject/3; hence we use if on line
5 instead of a while loop on the same condition).

6. Obtaining injection log. Similarly to calling inject/3 predicate, we call “getStrOut(OutChars).”
to obtain the buffer of characters, which represents the injection log. We use Term
t = ((Term) vars.get("OutS")); to retrieve the corresponding prolog term (array of
characters) and call t.getListChars() to get a Java String, representing the injection
log. This string is then added to the ConstraintState object.

7. Accumulating constraint. ConstraintState object is appended to the list of added con-
straints. This list is later used by generateKB() to generate the KB tree representation,
displayed in the front-end.

We have omitted details of the KB evaluator, implemented in Prolog, because they are
discussed later.

4.4.3 Extensions

There are two extensions to the generic injection process. Firstly, additional processing
is required for operation calls. Consider the following OCL example for our Car World
(Figure 2.2):

class Car

context Car::driveTo(x:Real, y:Real) : Boolean

body : x<50 and y<50
...

context Car

inv : self.driveTo(51, 50.5)

It specifies a contradictory set of constraints: invariants should always hold true, but in
this case the call to driveTo in the invariant results in false being returned by virtue of
definition, introduced in the body operation. How can we introduce the body definition into
the problem, so that our KB evaluator could derive the contradiction?
If we could produce an instantiated version of the operation body from its definition, the invariant would become $51 \leq 50$ and $50.5 \leq 50$, from which our KB evaluator can trivially derive the contradiction. This is implemented by processOpCall(constraint) method which is called by orderedInject() before the latter begins injecting the constraint into the KB evaluator. We have omitted some details when discussing the construction of semantic form: during CST traversal, for each post-condition and operation body we create an OpDeclNode (part of peterlev.semantics.analysis.node), which is used to carry attributes of the operation declaration, including the associated expression (which would be $\text{opArg1} \leq 50$ and $\text{opArg2} \leq 50$ for our example; recall that we re-name the arguments to the same common form). Operation calls are gathered during traversal as OpCallNode instances, which contain the operation call expression as well as instantiated arguments ($\text{numexp}(51)$ and $\text{numexp}(50.5)$ for our example). processOpCall() uses Java string manipulation functions (split(), replace()) and can be outlined as follows:

1. It determines whether the constraint contains operation calls by trying to match gathered OpCallNode instances against the constraint (using substring()).
2. It then finds the corresponding operation declaration (OpDeclNode) for each matched OpCallNode.
3. It uses the body attached to OpDeclNode to replace the operation call in the constraint. For our example, this produces \(\text{lt}(\text{varlet}(\text{opArg}), [\text{real}]), \text{numexp}(50))\) and \(\text{lt}(\text{varlet}(\text{opArg}), [\text{real}]), \text{numexp}(50))\).
4. Finally, “\text{varlet}(\text{opArg}, [\text{type}])” (argument references) expressions are replaced by their corresponding instantiations, which are retrieved from the OpCallNode ($\text{numexp}(51)$ and $\text{numexp}(50.5)$ for our example). This produces the complete semantic form $51 \leq 50$ and $50.5 \leq 50$ for our example.

Another extension is implemented to deal with operation bodies and post-conditions for operations, which have contradictory pre-conditions. As discussed above, body and post expressions should not be processed, if we had already encountered a contradictory pre-condition for the same operation. Pre-conditions, for which insertion into the KB failed, are collected in a list. This is handled inside orderedInject() by the following code:

```java
if(matchesFailedPre(constrActual)) {
    ConstraintState csTemp =
    addConstraintState(constr, false, false);
    csTemp.setSkipped(true);
    // constraint is skipped
    continue; // skips to next iteration
}
```

matchesFailedPre() retrieves a context of each failed pre-condition, and attempts to match it to a “[post, context]” or “[pre, context]” strings in the constraint (constrActual). A match means that the constraint to be inserted, is a post condition/body expression for operation, which should be skipped, since its pre-conditions are contradictory. Hence we insert the corresponding ConstraintState object for the constraint (line 3) and explicitly set it to denote a skipped constraint (line 4). We skip to the next iteration of the injection loop (line 6), which means the constraint does not get processed.

This completes all the extensions to the orderedInject() method. So far, we have omitted implementation details for KB evaluator, implemented in Prolog. We discuss the implementation in subsequent sections.
4.5 Evaluation of Constraints against the KB

We have illustrated the structure of KB evaluator in Figure 4.3. The main interface to the functionality is provided in main.pl. The main injection routine is defined by predicate inject/3. Ignoring routine implementation details, the routine consists of two main steps:

- constructing a list of relevant KB facts for the constraint, which is to be inserted. We describe this in greater detail further down.
- evaluating the constraint against this list of facts. This is performed by the evaluate(Constraint) unary predicate.

The evaluate/1 predicate performs the main function of the KB evaluator, and is specified in Prolog as follows (main.pl):

```prolog
evaluate(Exp)
    :- factlist(P), !,
       % fact list is already built in P
       printNegativeTest,
       \+ ( derive(P, neg(Exp)), printNegativeAns(true) ),
       % if deriving negation succeeds, proof failed printed
       % and derive/2 fails; OTHERWISE
       % KB does not imply the negation of Exp,
       % i.e. safe to add constraint
       printPositiveTest, derive(P, Exp), printPositiveAns,
       insertEx(true), printEx(Exp)
       % KB already implies Exp, i.e. excessive constraint
       ; % OR in prolog
       printNegativeAns(false) % prints not excessive msg
    !.
```

Firstly, it checks whether the constraint Exp is contradicted by the KB (relevant facts, already collected in list P) by attempting to derive the negation of constraint (line 4). If deriving the negation succeeds, then evaluate/1 fails (due to “\+”). Otherwise, the constraint satisfies the KB and there are two cases (represented by an OR on line 13): if we are able to derive the constraint explicitly using KB (line 10), then we flag the constraint as excessive (line 11) as it is already implied. Otherwise, evaluate/1 succeeds. Since this call occurs inside inject/3 of main.pl, if evaluate/1 succeeds, then the constraint is injected into the KB (using assert predicate in Prolog) inside inject/3. A cut (!) at the end, is used to ensure that the prolog interpreter never backtracks in an attempt to find another solution to evaluate/1, since there can only be one execution of this predicate.

To complete the injection routine, we still need to present how the relevant KB facts are built up in list P (which is retrieved on line 2) and how derive/2 evaluates the constraint against the supplied set of facts P.

4.5.1 Building List of Relevant KB Facts

All the constraints are asserted as dynamic facts of form constr/3 (semantic form for constraints). We use a routine buildfacts(Tag, Facts) to build up the relevant facts for a given “tag”. Tag determines the context of a constraint, e.g. invariants have tag “[inv]” since they apply to all states of the system, while, contextual element is included for other constraints,
e.g. \([\text{derive, } \text{attr} \text{(age, [carworld, car], [integer])}]\). \text{buildfacts/2} is specified as a typical recursive routine, which accumulates all constraints \(C\), that satisfy predicate \(\text{fact(Tag, C)}\). This predicate encodes the rules for building facts, which we specify below. Recall the semantic order for addition of constraints:

\[
def, \{\text{inv, init, derive}\}, \forall x (\text{pre}(x) \rightarrow \{\text{post}(x), \text{body}(x)\}).
\]

- **definition** is an assertion, which specifies helper a attribute/operation. Assertions therefore do not require any other facts for evaluation (also because they are inserted first by semantic order). They should be included for evaluation of any constraint.

- **invariant** should be evaluated against all other valid constraining facts in the KB, i.e. definitions (since helpers can be referred in invariants), initial and derived value expressions, since the latter represent constraints on attributes. Pre- and post conditions as well as operation bodies will never be added, since by semantic order they are processed after invariants. Definitions will not be added either, since all of them have already be processed in semantic order.

- **initial and derived values**; we have already mentioned that invariants and definitions should be added to the relevant fact list. We should clearly included other initial and derived values, which are for the same attribute. We use the context part of the tag (e.g. \([\text{init, attr}(...)]\)), to check for this.

- **pre-condition**; invariants and definitions should be included as mentioned. We should also include all derived value expressions, since they represent invariants for an attribute value. Initial value expressions, on the other hand, are not necessarily relevant: perhaps, the value of the attribute has changed since initialization of the system, and is different at the time the operation, for which pre-conditions are specified, is executed. This means initial values do not necessarily specify true facts, and should not be included in our relevant fact list. Post-conditions for this operation should not be included, since they only begin to hold after the operation is executed, i.e. not at the time of evaluation of pre-conditions for the operation. Post-conditions for other operations should not be included either, since they are not guaranteed to be true facts: e.g. post-conditions of operation \(y\) do not affect pre- and post- conditions of \(x\) if \(y\) is executed after \(x\). Since any operation body constraint can be specified equivalently as a post-condition, we do not include operation bodies either.

- **post-condition**; we use similar reasoning and hence only include invariants, definitions, all derived value expressions, post-conditions and operation bodies for the same operation os our post-condition. We do not include post- and body constraints for other operations, and also do not include pre-conditions for our operation. This is because in general the system state may change during the execution of an operation (e.g. thread) and we cannot guarantee that pre-conditions represent true facts at the time of evaluation of our post-condition.

- **operation body**; should involve exactly the same fact list as in the case of post-condition, which we have just described.

- **default case**; we have not explicitly mentioned that trivially for each constraint we should include all other constraints with exactly the same tag, e.g. for a pre-condition on some operation include other pre-conditions on the same operation in the relevant fact list.

The procedure for determining the relevant constraints, as specified above, is implemented in the \text{fact/2} predicate, which in turn is used for building up the list of relevant
facts. Once, it is built up by virtue of the rules, we have just described, the fact list \( P \) is asserted via `assert(factlist(P))`. Since this is done by `inject/3` before `evaluate/2` is called, the latter obtains the list of relevant KB facts by binding \( P \) as on line 2 of the Prolog code listing provided above.

### 4.5.2 Implementing Goal-Directed Prover

The heart of the semantic evaluation subsystem is our extended goal-directed prover. It implements the propositional goal-directed calculus, which we have presented in Section 2.3.

In addition, it has been extended to process numerical functions and relations.

First, we describe how the propositional goal-directed prover has been implemented in Prolog (module `derive.pl` of our KB evaluator). Recall, that two main procedures were involved: translation of logical formulae into Ready-For-Computation (RFC) form and applying an ordered set of rules on that form one-at-a-time until either the proof succeeds or fails. The main interface to our goal-directed prover is given by `derive/2` predicate which accepts a set of facts (which we will refer to as the data, which consists of only the relevant facts of the KB), and a goal (the expression we will be trying to derive):

```prolog
1 derive(Data, Goal)
2   :- !,
3      translate_list(Data, RFCData), % translate data.
4      translate(Goal, RFCGoal), % translates goal.
5      printRFC,
6      derive(RFCData, RFCGoal, [], []). % starts the proof of the RFC form.
```

Initially, we obtain RFC form for the data and the goal, and call `derive/4` predicate, which accepts data and goal in the RFC form, the goal history (set of all previous goals, we have attempted to prove within one call to `derive/2`), which is initially empty, and is required for the restart rule, and the rules history (set of important previously applied rules, which we use to prevent looping of our prover). Both histories are initially empty (line 6).

#### Implementing RFC Translation Rules

We exemplify implementation of RFC translation, which are based on recursive definition, provided in Figure 2.10. Predicate `translate/2` is implemented as a set of recursive clauses to accommodate this translation.

Consider rule (1) of the definition:

\[(A \rightarrow (B \land C)) \Rightarrow ((A \rightarrow B)^R \land (A \rightarrow C)^R).\]

(“\( R \)” indicates recursive steps, i.e. recursive calls to `translate/2`). It is implemented by the following clause of `translate/2`:

```prolog
1 translate(A implies (B and C), X and Y)
2   :- !, translate(A implies B, X),
3      translate(A implies C, Y).
```

Recursive calls result in computational sub-branches, which, when the base case is hit, produce the final form for each conjunct in \( X \) and \( Y \) (lines 2 and 3), which form the final output (\( X \) and \( Y \)). The base case for `translate/2` occurs when there are no more rules left to apply to the expression, i.e. when the latter does not match any clauses, it matches the most general mask “`translate(Exp, Exp)`.” which simply leaves the expression unchanged,
since it is guaranteed to be in RFC. In the context of Figure 2.10, if none of the 9 rules can be applied to the expression, it has reached the RFC form (which we illustrated with a translation example in Section 2.3). Other clauses of translate/2 are implemented in a similar fashion, and result in recursive call on a simplified form of the expression. Since at each stage, the form is simplified, translate/2 routine is always guaranteed to terminate in a finite number of steps. translate_list/2 is a predicate, which calls translate recursively on each member of the supplied Data and produces a list, each member of which is in RFC form.

Implementing Goal-Directed Rules

As shown in the Prolog code listing for derive/2 above, once RFC form is obtained for the Data and the Goal, we call the theorem prover itself via derive/4. Each clause of derive/4 implements a goal-directed rule, all of which have been presented and exemplified in Section 2.3.

We exemplify implementation of Atomic Rule (Implied Goal), which states “reduce $P, A \rightarrow G \vdash G$ to showing $P, A \rightarrow G \vdash A$”. The corresponding clause of derive/4 is given below:

```
1  derive(Data, B, History, Prev)
2  :- primitive(B), member(A implies C, Data),
3      proves([C | Data], B, RF), !,
4      printstep(Data, B, History, ruleImpliedGoal), % print method
5      delete(Data, A implies C, DataN), % removes A->C from data
6      simplifyData([C | DataN], DataS),
7      printProof(DataS, RF, B, History, 'direct inference'),
8      add(B, History, HistoryN), % add B to goal history
9      derive(DataN, A, HistoryN, Prev). % to prove A.
```

We begin by verifying whether the rule can be applied, i.e. B is an atomic goal (satisfies primitive(B)) and there is some $(A \implies C)$ in the Data, such that C matches B (checked using proves/3). Then we print that the rule is being applied (line 4). The necessary modifications to the Data involve removing $(A \implies C)$ since once this rule is applied, the implication will not introduce any extra semantics anymore. We omit calls on lines 6 and 7 for the time being. Finally, since B is an atomic goal, it is added to the history of previous goals (line 8), and the theorem prover (derive/4) continues with the updated data and goal history, trying to prove A. Several points remain unclear:

- why have we defined our own predicate primitive/1 for testing if an expression is atomic, if that could be done using the built-in atom/1 predicate? Just for propositional logic, atom/1 is sufficient, but it becomes insufficient for our extensions to the theorem prover, which will be presented later. Therefore, at this stage primitive(B) is effectively defined as atom(B), but the former will be extended.

- why not match B directly via variable binding using member(A implies B, Data) instead (in line 2), i.e. what is the point in calling proves([C — Data], B, RF)? Similar to the point above, it is true that direct matching is sufficient for propositional logic, e.g. atom a matches a. But for our extensions to the prover, this notion of “matching” needs to be generalised. The generalised semantics is provided by proves/3, which succeeds the given list of facts “matches” the goal in a general sense (information about this matching, which is required for printing is returned in RF). We define this generalised semantics in the discussion of our extensions.
The prover we have presented is able to operate on atoms, e.g. $a$, $b$, (which is useless in the context of OCL) and built-in OCL type Boolean, since the latter directly corresponds to logical $\{\top, \bot\}$. However, Integers, Reals and Strings are not supported: we are not able to deal with expressions such as, $a > 5$, $b = c$, ‘peter’ $'$='PETER’ etc. Hence, we still need to implement facilities for representing types Integer, Real and String in our prover and implement support for operations on these types (relations $>$, $<$ etc, functions $+$, $-$, $*$, $/$ etc).

### 4.5.3 Extensions for Numerical Processing

The theorem prover which we have exemplified up until this point is powerful enough to deal with propositional logic. Moreover, if we assume that our set of Prolog predicates reproduces the exact semantics of goal-directed calculus, then our automated theorem prover is sound and complete for propositional logic, since the calculus itself is sound and complete (discussed in Section 2.3).

However, on atomic level, the system is insufficient to deal with anything more complex than, say, “a” (name any constant you like), i.e. it is not powerful enough to reason on OCL constraints, which we plug into the KB evaluator in semantic form. We ignore collection types, which leaves us with absence of support for types, such as Integer and Real in OCL, including their associated pre-defined operations (e.g. $>$ on type Real).

There are new notions we have to accommodate:

- atoms could also be numeric relations, e.g. $a > b$ is a logical atom, since it cannot be decomposed further in propositional logic.
- the notion of matching atoms is more general, e.g. $a > b$ should “match” $a > c$ if $b > c$. This cannot be simplified further using our goal-directed prover, since all the elements involved are atomic in propositional logic.

To implement support for these types, we extend predicates proves/3 and primitive/1. *Extensions for primitive/1* are trivial: we simply have to specify explicitly what kinds of expressions can be atomic:

```prolog
clause "primitive(eq(_,_))." specifies that numerical relation “equals” can be atomic.
```

We insert similar clauses for all possible atomic facts. Note that the only restriction is that they have to evaluate to type Boolean (as facts), hence numerical functions, e.g. `plus(_,_)`, cannot be atomic, and only appears as an argument to numerical relations such as, $=, =<, =>$, $=, =<, =>$. We will refer to this more general notion of an atomic fact as a *primitive*. Hence, `eq(numexp(5), numexp(4))` is a primitive ($5 > 4$).

Predicate proves/3 currently supports only simple matching, but as mentioned, it needs to be extended:

```prolog
1   proves(Data, B, '')
2      :- primitive(B), member(B, Data), !.
3
4   proves(Data, ArithExp, ReducedForm)
5      :- simplifyData(Data, DataS),
6          evalArith(DataS, ArithExp, ReducedForm, truth).
```

The first clause of proves/3 (lines 1 and 2) is the original definition, which handles simple matching: if for a goal B, B is already in Data, then we succeed. We introduce a more general notion: all logical connectives are removed from Data (line 5), so that only primitives are kept. The latter are then passed to evalArith/4 predicate, which performs numerical
evaluation on the primitives, to establish whether primitive goal ArithExp can be derived. 
evalArith/4 is the main interface to mathcall.pl which performs numerical evaluation via Mathematica (this is discussed later). Recall, our Prolog code listing for derive/4 above: we have just presented the definition of proves/3, which is called from derive/4 (on line 3) to establish whether the goal B could be derived directly from the Data, augmented with C, which is the conjunct of an implication in the Data. We have thus defined a more general notion of “C matches B”, which we will refer to as direct inference. It is a proof procedure, just like our goal-directed theorem prover, with the difference between them being that direct inference involves only numerical relations, and no logical connectives. Our goal-directed prover, on the other hand, can only reduce logical connectives, and considers numerical relations as atomic, irreducible facts (primitives).

This seems to provide a complete framework for evaluating logical expressions, which can also involve numerical relations and functions. However, consider this example: show $a > 6 \vdash \neg(a \leq 6)$. This causes our goal-directed prover to fail, since there are no rules to apply. Yet, trivially, the proof should succeed: negation of $a \leq 6$ gives $a > 6$, which is given in the data. The notion of “excluded middle” for numerical relations is still lacking in the prover, i.e. either relation $=op=$ is true, or its complement $=cop=$ is true (formulation of $A \lor \neg A$ for numerical relations). The complement of $>$ is $\leq$, complement of $\geq$ is $<$ and so on. We specify this in our system using predicate arithconverse/2, which consists of a set of facts: e.g. “arithconverse(eq(A, B), le(A, B)).” takes $A = B$ as an argument, and returns its complement in $A \leq B$.

We also still require a way of inserting this notion of excluded middle into our goal-directed prover. Note, that this notion of “excluded middle” cannot be inserted in general form, which will hold for all relations, since any non-logical expression in propositional logic is treated as atomic. Hence, we propose the following additional rule to our goal-directed calculus, called the

**Law of Excluded Middle (Numerical Relations)**

\[
given P \vdash C, \text{where } (A =op= B) \in P, \text{ show } P \cup \{(A =cop= B) \rightarrow \bot\} \vdash C \text{ (where } =op= \text{ is a numerical relation, and } =cop= \text{ is its complement as given by arithconverse/2).}
\]

Note that, formally speaking, this is not a goal-directed rule. In our goal-directed evaluation order, this rule should be applied only if all the preceding rules (including restart rule) fail. The rule essentially “helps” the prover, by introducing the notion of “excluded middle” for the first numerical relation, found in the data, as an extra fact. The next rule that will be attempted is always Atomic Goal (Implied Bottom) (there is an implication of falsity in the data, hence attempt to prove its antecedent $(A =cop= B)$).

We implement this rule as an additional clause of derive/4 as follows:

```
1  derive(Data, A, History, Prev)
2    :- member(B, Data), primitive(B), isArithOp(B), !,
3       arithconverse(B, BConv),
4       non_member(BConv implies boolean(false), Prev),
5       printstep(Data, A, History, ruleArith),
6       add(BConv implies boolean(false), Data, DataN),
7       add(BConv implies boolean(false), Prev, PrevN),
8    !, derive(DataN, A, History, PrevN).
```

First, we check whether pre-conditions for this rule are satisfied (line 2): there is an numerical relation B in the Data, which has a complement BConv. We obtain the arithmetic complement of B in BConv (line 3), print the applied rule (line 5) to the proof log (which becomes part of the constraint injection log) and insert $BConv \rightarrow \bot$ into the Data (line 6).
Note that we also insert this into the history of rules, to make sure that we do not loop. Otherwise looping can occur as follows:

- Nothing left to try for \( a \geq 5 \vdash c \). Apply our rule to the first numerical relation in the data, \((a \geq 5)\).
- We obtain \( a \geq 5, a < 5 \rightarrow \bot \vdash c \). Atomic Rule (Implied Bottom) is applied next.
- \( a \geq 5 \vdash a < 5 \). Nothing left to try. Hence we apply our rule again for \((a \geq 5)\), which results in a loop. This is why we maintain a history of previous instructions. Note that one of the pre-conditions to our derive/4 clause above checks whether we have already tried inserting the fact (line 4). Hence, at this point we should fail the proof (because no valid proof exists).

A cut before the recursive call (line 8), is added to ensure that Prolog does not backtrack to find other solutions, if our recursive call fails.

Adding this derive/4 clause right before the last clause (which is the clause “proof fails: no other rules to try”) completes our implementation of the extended theorem prover.

### 4.5.4 Numerical Evaluation using Mathematica

evalArith/4 predicate, which we have mentioned above (called by proves/3) is used to infer whether as set of provided numerical relations implies the given goal. We refer to this as direct inference, e.g. \( \{a > 5, a = b\} \) directly derives \( b > 4 \), since \( b > 5 \rightarrow b > 4 \). evalArith/4 is specified as part of mathcall.pl module, since it employs Mathematica to establish direct inference (as illustrated in Figure 4.2).

In particular we use the following sequence of routines (which we will refer to as Reduce-Simplify) provided by Mathematica:

1. \( \text{Reduce}[	ext{Data} \cup \text{DomainConstraints}, \text{Variables}] \), which reduces a set of constraints and inequalities (first argument) with respect to specified variables (second argument). The advantage of using Reduce is that it comprises several powerful Mathematica routines, which it attempts automatically in order to find an optimal reduced form RF. It is worth to note, that Reduce is a reversible operation, since the output RF represents the same mathematical set as provided to Reduce, hence no semantics is lost. We append domain constraints for each variable to the data: e.g. if for \( x > 5 \) we know that \( x \) is an integer, the resulting argument passed to Reduce is \( \text{Greater}[x, 5] \&\& \text{Element}[x, \text{Integers}] \) (set \( \{x > 5, x \in \text{Integers}\} \).

2. \( \text{Simplify}[	ext{Goal}, \text{RF}] \) performs a sequence of transformations on Goal, given assumptions RF to find the simplest form of Goal. We have already inferred new semantics from original data via Reduce, which produces RF. If we can simplify Goal to True, assuming RF, then this means that \( \text{Data} \rightarrow \text{Goal} \).

An example of this sequence of calls is given in Figure 4.4 (this would result from a typical call to evalArith/4, in an attempt to show \( a = 6 \) by direct inference).

In essence, mathcall.pl provides a set of predicates to convert all expressions in Data (in Prolog form) to their according Mathematica forms. This produces DataM, which is then augmented with a set of domain constraints, (generated in list DomL via a call to predicate addDomConstraints(DomL) from mathcall.pl). All the variables, which appear in
Figure 4.4: Example of the Evaluation Routine in Mathematica

Data, are collected into a set Variables. We thus construct a Mathematica call as a String “Reduce [...]”. Jasper (Prolog→Java) interface, is used to call `evalExpr("Reduce[...]")` on an instance of MathCall (recall that our KB evaluator can access the reference to this object, since it was explicitly passed by ConstraintInjector). `evalExpr()` returns the answer string, which is the reduced form, produced by Mathematica. In Prolog, we do not care what the string represents, as it is fed as-is into a `Simplify[...]` call to Mathematica, with the Goal also converted to Mathematica form from semantic form. If `evalExpr("Simplify[...]")` returns “True”, then direct inference succeeds, and hence `evalArith/4` succeeds. In any other case, it fails.

Conversion to Mathematica Form

Conversion of semantic form to Mathematica form is trivial in most cases, because Mathematica has corresponding built-in types for Integers, Reals, Booleans and Strings e.g. prolog fact `numexp(5)` is converted to string (array of character codes in Prolog) “5”, because the entire command is passed to Mathematica as a string. We can exemplify some other conversions:

- `boolean(true)` and `boolean(false)` are converted to “True” and “False” correspondingly.
- `gt(plus(numexp(5), numexp(6)), minus(numexp(5), numexp(4)))` (5 + 6 > 5 – 4) is converted to `Greater[Plus[5,6], Subtract[5,4]]` in Mathematica (which is stored in Prolog as a string).
- `remain(numexp(5), numexp(2))` is converted to `Quotient[5,2]`. As can be seen, using Mathematica is extremely convenient, since it already has built-in functions to manage all the built-in operations of OCL on Integers.

Special cases involve converting UML attributes and operations from their semantic form, since these are not supported in Mathematica explicitly. Instead we construct Mathematica variables, in such a way, that the variable name uniquely represents the given attribute/operation:

- Attributes are uniquely specified by their name, context and type. Hence, e.g. `attr(age, [carworld, car], [integer])` (`carworld::car::age : Integer in OCL`) is converted to Mathematica variable name “car-worldDOTcarDOTageSCOLinteger”. We use insert keywords “DOT” and “SCOL”,

85
so that when output from Mathematica is received, we can convert the variable name back to the semantic form for the attribute *age*, i.e. conversion is reversible.

- **Operation signatures** are uniquely specified by their name, context, and an ordered set of argument types, that the operation accepts. E.g. \(\text{oper(driveTo, [carworld, car], [[real], [real]], [boolean])}\) (\(\text{carworld::car::driveTo(x:Real,y:Real):Boolean}\) in OCL) is converted to Mathematica variable name “\(\text{carworldDOTcarDOTdriveToSCOLrealCMArealAEND}\)”. Similarly, keywords “DOT” are used to delimit the context of the operation. “SCOL” indicates the start of argument types, which are delimited by “CMA”, and terminated by “AEND”. Hence, the variable string can be used to re-construct the original semantic form for the operation.

- **Operation calls** are processed in exactly the same way, except instead of argument types, we specify the argument as a “CMA”-delimited string: e.g. for a call to \(\text{driveTo(4, 4.5)}\), the string given for the operation signature above would instead end with “...SCOL5CMA4TOD5AEND”. Note that “4.5” represents a tricky case, since we cannot simply convert it as a usual \(\text{numexp(4.5)}\) (dots not allowed in variable names in Mathematica). In cases like these, “.” is replaced by “TOD” keyword. Hence, each operation call corresponds to a unique variable. The same operation calls are denoted by the same variable names, hence semantics of expressions in Mathematica is not affected.

In *mathcall.pl*, the conversion rules which we have described, are defined as clauses of the \text{opToMath} predicate. Conversion of Strings and associated built-in OCL operations to Mathematica form is trivial: = and \(<>\) relations work the same way as with arithmetic, Mathematica itself provides functions for Strings, which can represent OCL operations on strings (e.g. \(\text{s.substring(1, 5)}\) in OCL is represented in Mathematica as \(\text{StringTake[s, \{1,5\}]}\)). Conversion procedure is the same as for arithmetic, and is implemented by additional \text{opToMath} clauses in *mathcall.pl*.

### 4.6 Conclusions

We have thus presented implementation details of our semantic evaluation subsystem, which is illustrated in Figure 4.2. It performs injection of constraints into the Knowledge Base using the KB Evaluator implemented in SICStus Prolog, and returns results and constraint injection logs in human-readable format back to the Java front-end.

At the core of the system lies the Knowledge Base Evaluator, which is based on a propositional goal-directed calculus. We have presented an extended implementation of this calculus, which also provides support for numerical relations. The extended theorem prover we developed, uses Wolfram Mathematica to perform numerical evaluation via the \text{Reduce-Simplify} routine. The routine also involves invertible conversion procedures, which produce Mathematica form for expression in internal Prolog semantic form.

In addition to the evaluator, the system implements a sophisticated printing module *print.pl*, which produces human-readable form, which is ultimately passed back to the front-end and displayed to the user.

In subsequent chapters we demonstrate the flexibility and functionality of the system, discuss and evaluate our design choices and consider future extensions to the system.
Chapter 5

Testing: Case Study

In this chapter, we present a comprehensive case study, which has been used to test our OCL Evaluation Framework. The case study consists of:

- a UML class model, which we will be testing our OCL constraints against
- a test file with OCL constraints, which will be used to test syntactic evaluation subsystem, implemented in OCL Evaluation Framework
- a test file with OCL constraints for testing semantic evaluation subsystem

5.1 Testing Environment

Current version of OCL Evaluation Framework requires SICStus Prolog to be installed, and needs to access the supplied Mathematica 6 kernel (via OS environment variable). Since the purpose of this initial release is demonstration only, we have not provided any stand-alone distributions with automatic installation.

We performed tests on OCL Evaluation Framework, which has been executed in the following environment:

- **Operating System**: Microsoft Windows XP SP3 32-bit / Windows Vista SP1 32-bit
- **CPU**: Intel Mobile Core 2 Duo T7250 @ 2.00 GHz
- **RAM**: Dual DDR2 SDRAM 3.00 GB
- **Additional Software**:
  - Sun Java: JRE version 1.5.0 build 14 (located in JAVA_HOME)
  - SICStus Prolog: version 3.12.5; includes Jasper (located in PROLOG_HOME)
  - Wolfram Mathematica 6: kernel and minimal required packages, version 6; includes JLink
- **OS Environment Variables**:
  - PATH = %JAVA_HOME%\bin;%PROLOG_HOME%\bin
  - JAVA_HOME = %JAVA_HOME%

We begin by presenting the UML class model used for testing.
5.2 UML Class Model

The main priority in designing the underlying UML class model was richness of description: the class diagram must involve all supported model elements, which can meaningfully be referred to from OCL.

We have already presented several examples from the case study throughout this text. The UML class model we will be using is the Car World example in Figure 5.1. The model can be loaded as an XMI file into our OCL Evaluation Framework. It models a “Car World”, where each person (classifier “Person”) can own several cars (classifier “Car”) via association “Ownership”. For each car a person purchases, there is a base price (attribute “price” inside association class “Purchase”), the actual price paid (attribute “purchasePrice”), and a function, which computes the purchase price from the base price of the car (operation “computePrice()”). Each car also has several characteristics: the model name (attribute “model”), the date of production of the car (modeled by attribute “productionDate”) and the maximum speed it can reach (attribute “topSpeed”). The latter can also be accessed via getter (operation “getTopSpeed()”), and setter (operation “setTopSpeed(double)”) functions. In addition, the car’s location can be changed using operation “driveTo(Real, Real):Boolean”, which takes coordinates of the new location, and returns true if moving the car has been successful, and false otherwise (in this case, the location is already occupied by another car). Each car is a result of production (association “Production”) by one of the car manufacturers (classifier “Producer”) in our Car World. Each producer is characterized by its name (attribute “name”).

This provides a complete specification of our Car World as a model. It includes all main
model elements: classifiers, attributes, operations, associations and association classes, and therefore can facilitate a multitude of highly expressive constraints, which we will define in OCL.

We begin with testing of our syntactic evaluation subsystem.

5.3 Syntactic Evaluation Testing

Recall that our syntactic evaluation subsystem provides two main checks: syntax parsing, which ensures that the specified constraints conform to the OCL syntax, and contextual evaluation, which verifies that UML model elements are navigated and referred correctly, types of expressions are satisfied, operations are called correctly (we refer to these checks as contextual evaluation).

Contextual evaluation can only be performed if syntax parsing has succeeded. Therefore, the our OCL constraints for testing contextual evaluation, test syntax parsing as well. We have defined a comprehensive test file with OCL constraints, which test every major aspect of the OCL syntax. We will illustrate some of the interesting and more complex constraints in this chapter.

Since we have tested semantic subsystem separately, at this point we are not concerned with satisfiability of constraints. Since syntax parsing and contextual evaluation are a form of yes/no testing (either expressions are verified correctly or otherwise), we exhibit constraints, that our system has both, parsed and contextually evaluated, correctly.

We begin with some invariants, which should always hold true for cars (classifier “Car”) in our Car World:

context c:Car

inv nameGiven : c.model <> 'unnamed'

inv noOldCars : productionDate.substring(productionDate.size()-4, productionDate).toInteger() > 1980

inv fastSpeed : c.topSpeed > -100

inv speedOk : self.topSpeed < 400 and (self.topSpeed < 0 implies false) = topSpeed > 0

We have assigned names to all invariants. “nameGiven” simply specifies that all cars must be named (later, we specify that “unnamed” is the value of attribute model at initialisation). “noOldCars” uses built-in OCL operation substring() operation on String to retrieve the year of production of the car (we assume dates are given in format “dd/mm/yyyy”), and asserts that all cars in our Car World are newer than 1980. “fastSpeed” asserts that all cars in our Car World are relatively fast and can achieve speeds beyond 100 kms/h: we take -100 (integer), and explicitly force the Real type, which then enables us to call round() (not defined on Integer), and then take the absolute value, which gives 100. Finally, “speedOk” gives bounds for speed limits in all cars of our Car World. We specify the upper bound as 400 kms/h. Note that the second conjunct does not specify that the topSpeed ¿ 0, it merely expresses that ¬(topSpeed < 0) = topSpeed > 0, which holds for any value of attribute topSpeed.

The specified invariants test context naming (one of our extensions to Dresden OCLParser), some String and Real operations, OCL type conformance operation oclAsType, logical con-
nectives, and numerical relations > and =.

We specify some initial and derived value expressions for attributes:

```
context Car::model : String
  init: 'unnamed' --see inv "nameGiven"

context Person::age : Integer
  derive: if ownedCar->notEmpty()
          then 18
          else self.age@pre
          endif

context Purchase::purchasePrice : Integer
  init: 0
  derive: if self.price >= 1000
         then self.computePrice()
         else price
         endif
```

All model names are initialised to “unnamed”. If a person owns a car, then the minimum feasible age must be at least 18, and is hence set to 18 in the records of our Car World. “@pre” is tested, and indicates that if no car is owned, the record should not be updated (i.e. the value remains the same as it was). We also test if-then-else construct throughout these expressions. Purchase price of cars is initialised to 0. However, if the base price (attribute price) reaches 1000, then we use a pre-defined function to compute the purchase price. Otherwise, purchase price is the same as the base price. Note that we test type checking and reference of contextual classifiers extensively. “age” and “purchasePrice” are correctly evaluated as constrained elements, while association class “Purchase” and classifiers “Person” and “Car” form constrained contexts. Recall that, association classes are one of our extensions to the OCLParser.

We define some helper operations/attributes, which we will use in other OCL constraints. At the same time, we are testing OCL definitions, which are one of our extensions, implemented in the syntactic evaluation subsystem in OCL Evaluation Framework:

```
context Car
  def: attr locX:Real = 0.oclAsType(Real)
  def: attr locY:Real = 0.0

  def: oper isSpaceFree(x:Real, y:Real) : Boolean
       = Car.allInstances()->
         select(c:Car | c.locX = x and c.locY = y)->isEmpty()

  def: oper validPos(x:Real, y:Real) : Boolean
       = x > 0.0 and x < 50.0 and y > 0.0 and y < 50.0

inv oneCar: Car.allInstances()->forall(c1:Car |
     validPos(c1.locX, c1.locY) implies
     not Car.allInstances()->exists(c2:Car |
           c2.locX = c1.locX and c2.locY = c1.locY))
```
We have defined additional attributes to store current location of each car (both coordinates initialised to 0, which is outside the board; hence initially, no cars are on the board in our Car World) in \( \text{locX} \) and \( \text{locY} \). In addition we have specified a predicate \( \text{isSpaceFree()} \) which takes \( x \) and \( y \) coordinates of a space on the board, and returns true if the space is not occupied, and false otherwise. The operation is specified elegantly using powerful allInstances() operation, one of the critical extensions, which we have provided in OCL Evaluation Framework. We retrieve all cars in the Car World, and obtain a subset of cars, which are in the same location, as specified by arguments \( x \) and \( y \) (using our attribute helpers to denote location of each car). If this subset is empty, we return true (i.e. no other cars in that location). We also specify a simple helper operation \( \text{validPos()} \), which indicates whether given coordinates specify a valid position on the board.

Invariant “oneCar” specifies that each space can be occupied by at most one car. In logical notation, the invariant expresses:

\[
\forall c_1 (c_1 \in \text{Car.allInstances()} \land \text{validPos}(c_1.\text{locX}, c_1.\text{locY}) \rightarrow \\
\neg \exists c_2 (c_2 \in \text{Car.allInstances()} \land c_2.\text{locX} = c_1.\text{locX} \land c_2.\text{locY} = c_1.\text{locY}))
\]

It makes heavy use of collection operations to specify that for each car on the board \( (\text{validPos()}) \) is true), there is no other car which is in the same position on the board. Note that we can specify this by referring to each valid position as opposed to each car:

\[
\text{inv: let } s1: \text{Sequence (Integer)} = \text{Sequence \{ 1..50 \} in} \\
\text{s1}->\text{forall (x: Integer | s1}->\text{forall (y: Integer | validPos(x, y) implies} \\
\text{let count: Integer = Car.allInstances() ->} \\
\text{select (locX = x and locY = y)->size()} \\
\text{in} \\
\text{(count = 1 or count = 0))}
\]

Unfortunately, the constraint only works for integer positions, since OCL does not yet allow one to specify either free variables or infinite collections. The invariant instantiates a sequence of integers 1.2..50 (which we will use as a set of all possible coordinates). For all coordinates \( x \) and \( y \), which specify a valid position on the board, either there is exactly 1 car in that position or there are no cars in that position. We express this by obtaining the size of the subset of all cars, which are in the same location as \( (x,y) \), in variable \( \text{count} \) and asserting \( \text{count} = \{0,1\} \).

The two invariants provide a comprehensive test of built-in OCL Collections and operations such as \( \text{select, forAll and exists} \) defined on them. Finally, we illustrate some tests of pre-, post- and body conditions which have been used in the case study to test our syntactic evaluation subsystem. We will refer to previously defined helper attributes and operations.

\[
\text{context Car::driveTo(x:Real, y:Real) : Boolean} \\
\text{pre argsOk : validPos(x, y) } \\
\text{body : isSpaceFree(x,y)}
\]

\[
\text{post : if } \text{isSpaceFree(x,y)} \\
\text{then self.locX = x and self.locY = y } \\
\text{else locX = locX@pre and locY = locY@pre } \\
\text{endif}
\]

\[
\text{context Car::isSpaceFree(x:Real, y:Real) : Boolean} \\
\text{pre : validPos(x,y)}
\]
\[ \text{post: } \text{locX} = \text{locX@pre and locY} = \text{locY@pre} \]

context Purchase::computePrice() : Integer
\[
\text{post: result} = ((0.80*price/3).\text{round}()-3*price).\text{abs}()+price
\]

The pre-conditions for \text{driveTo()} specify that arguments must define a valid position on the board. Note that body constraint specifies that the operation returns \text{isSpaceFree()} (i.e. true if the space is free). “post: isSpaceFree(x,y)” would not be a valid specification, since after \text{driveTo()} is executed, the space becomes occupied. Indeed, this is asserted by the post-condition we have specified above. If the space is not free, position of the car is kept unchanged (specified using \text{@pre}). We also test that pre- and post- conditions can be specified on helper operations, which are not part of UML and which were defined in OCL using \text{def}. For \text{isSpaceFree()}, the post condition effectively specifies that it is a \text{query operation}, i.e. it does not modify internal state of the system. Finally, we specify the return value of \text{computePrice} as a complicated arithmetic function, for the purpose of illustrating one of our test of arithmetic type checking in OCL Evaluation Framework. The expression effectively specifies: \[ \text{result} = |\text{round}(0.80 \times \text{price} ÷ 3) - 3 \times \text{price}| + \text{price}. \]

Recall that we have exemplified only a fraction of constraints, which have been tested with OCL Evaluation Framework.

### 5.4 Semantic Evaluation Testing

In the case of semantic evaluation, the test set of OCL constraints has two cover three main aspects:

- \text{richness of logical connectives} in constraints, is required in order to test the ability of semantic evaluator to resolve logical expressions
- \text{numerical relations} should constrain the domain of feasible solutions in a way that produces non-straightforward answer sets. This constitutes a good test for evaluating the component of our system, which deals with semantics of Integers, Reals and Strings and supported operations, e.g. \text{mod()} etc
- \text{relevant facts specified using different constraint types}, to evaluate the extent, to which our semantic evaluator is able to combine knowledge and infer semantics from different constraint types

We will again exemplify only a fraction of test cases used. The following constraint defines a trivial assertion about car purchases in our Car World: if the purchase price is 1000 then \text{computePrice()} operation also returns a 1000. If the stated holds, then our specification of \text{computePrice()} is correct and hence the purchase price must be at least 100. If the entire implication holds, then the purchase price is definitely positive.

context \text{p:Purchase}
\[
\text{inv: } ((\text{self.purchasePrice} = 1000 \text{ implies p.computePrice()} = 1000)
\text{ implies self.purchasePrice} >= 100
) \text{ implies p.purchasePrice} > 0
\]

In propositional logic, this would correspond to a tautology \((A \rightarrow B) \rightarrow A \rightarrow A\), i.e. it always holds (irrespective of true facts in the system). Note that we chose \text{A} to represent three expressions simultaneously: \text{purchasePrice} = 1000, \text{purchasePrice} >= 100 and \text{purchasePrice} > 0. This is due to
the fact that by properties of numerical relations, each of the three expressions derives its right-handed neighbors, the notion which we have previously referred to as direct inference. Therefore our semantic evaluator should deduce that:

- the constraint satisfies our empty KB (tautologies satisfy any KB);
- constraint is excessive, in that it does not introduce any new semantics to the KB (this is because we can deduce the constraint with an empty KB, i.e. we require no additional semantics to prove it);

Can we prove that the constraint is a tautology? To understand the proof, one can analyze the constraint injection log, produced by our KB evaluator (screenshot of the KB evaluator for this example is given in Figure 5.2). The constraint node in the Knowledge Base tree is highlighted in yellow, which indicates that the former is excessive. Recall, that the KB evaluator attempts to derive the negation of the constraint. If that succeeds, the constraint contradicts the KB. Otherwise, it satisfies the KB, and the evaluator then attempts to derive the constraint given the KB facts to check whether it is already implied. In our case the KB is empty, and the structure of the constraint injection log can be outlined as follows:

1. **attempt to derive negation of constraint** (our KB is empty, hence initial Data is empty):
   \[
   [] \vdash (((\text{purchasePrice} = 1000 \rightarrow \text{computePrice}() = 1000) \rightarrow \text{purchasePrice} \geq 100) \rightarrow \text{purchasePrice} > 0) \rightarrow \bot.
   \]
2. apply *rule for implication*; we insert the antecedent into the Data, and attempt to prove the consequent (⊥ in our case) with the new Data:

\[
\left( (\text{purchasePrice} = 1000 \rightarrow \text{computePrice}() = 1000) \rightarrow \text{purchasePrice} \geq 100 \right) \rightarrow \text{purchasePrice} > 0 \vdash \bot.
\]

3. no rules left to apply. proof fails, i.e. the constraint does not contradict the KB.

4. attempt to derive constraint using the relevant KB facts as Data (checks whether constraint is *excessive*). KB is empty, hence initial Data is empty:

\[
[] \vdash (\text{purchasePrice} = 1000 \rightarrow \text{computePrice}() = 1000) \rightarrow \text{purchasePrice} \geq 100.
\]

5. apply *rule for implication*; we insert the antecedent into the Data, and attempt to prove the consequent (⊥ in our case) with the new Data:

\[
(\text{purchasePrice} = 1000 \rightarrow \text{computePrice}() = 1000) \rightarrow \text{purchasePrice} \geq 100 \vdash \text{purchasePrice} > 0.
\]

6. apply Atomic Rule (Implied Goal): we have some implication \( A \rightarrow B \) in Data, such that \( B \) directly derives the current goal. We remove the implication from Data, and attempt to show antecedent \( A \), since the latter implies the current goal (note that we also insert the current goal into the history of goals). In our case, \( \text{purchasePrice} \geq 100 \) is the consequent of an implication, and it derives \( \text{purchasePrice} > 0 \) directly (trivial). Hence, we attempt to prove the antecedent of the implication:

\[
[] \vdash \text{purchasePrice} = 1000 \rightarrow \text{computePrice}() = 1000.
\]

7. apply *rule for implication*; we insert the antecedent into the Data, and attempt to prove the consequent (⊥ in our case) with the new Data:

\[
\text{purchasePrice} = 1000 \vdash \text{computePrice}() = 1000
\]

8. apply Law of Excluded Middle (Numerical Relations). We insert a new fact into the Data, which specifies that the complement of \( \text{purchasePrice} = 1000 \) is false:

\[
(\text{purchasePrice} = 1000, \text{purchasePrice} <> 1000) \vdash \text{computePrice}() = 1000
\]

9. apply Atomic Rule (Implied Bottom). ⊥ is a consequent of an implication in Data, if we prove its antecedent, then we can prove anything, i.e. our current goal (note that we remove the implication from Data, and add the goal to our history of goals):

\[
(\text{purchasePrice} = 1000) \vdash \text{purchasePrice} <> 1000
\]

10. apply Restart Rule. We reset the goal to one of our previous goals, which is kept in goal history. We changed this goal in step 6: \( \text{purchasePrice} = 1000 \vdash \text{purchasePrice} > 0 \)

11. proof succeeds. Goal is directly derived by the data (trivial). Hence, the constraint is already implied by the KB, i.e. it is excessive, and should not be added to the KB.

We have thus provided a complete sequence of steps, which our KB evaluator performs for this example. Proofs details become significantly large and tedious as examples get more complex and larger. Hence, we will outline only the intuition behinds proofs in future examples, and omit complete proof details. Full constraint injection logs can be re-produced by executing examples in OCL Evaluation Framework.

The following constraints test numerical evaluation, by specifying contradictory pre-conditions:

```ocl
class Car {
  def driveTo(x:Real, y:Real) : Boolean {
    pre: let c:Integer = 2 in
    return ...
  }
}
```
\[ x > c \text{ and } (x+6 \geq y \implies y \geq 4 \text{ and } x =< 6) \]

context Car::driveTo(a1:Real, a2:Real) : Boolean
pre: a1 > a2 and a2 >= 6

Contradiction is not immediately obvious here. The first pre-condition initially specifies \(x \in [2, +\infty)\) in the first conjunct, but then the implication gives \(x \geq y \implies (y \in [4, 12] \land x \in [4, 6])\). Although the second pre-condition is specified on the same operation, different argument names mask \(x\) and \(y\) as \(a1\) and \(a2\) correspondingly. The first conjunct gives \(x > y\), which reduces the set of possible solutions to \(y \in [4, 6]\), since \(x\) can be at most 6. But the second conjunct implies \(y \in [6, +\infty)\). Formally, \([4, 6] \cap [6, +\infty] = \emptyset\), i.e. we have derived a contradiction. Figure 5.3 shows the screenshot of semantic evaluation of this example, which

![Screenshot of semantic evaluation](image)

Figure 5.3: Screenshot of the KB Evaluator for the Numerical Contradiction Example

has succeeded in OCL Evaluation Framework. The example tests the ability of semantic evaluator to interpret let-in expressions, match operation signatures and hence combine their pre-conditions. Also, our evaluator has shown that it can determine solution sets for variables involved in numerical relations, which are in turn combined with logical connectives.

The first pre-condition is not itself contradictory and is therefore inserted into the KB successfully, since the latter is initially empty. However, the theorem prover is able to derive the negation of the second pre-condition, which means it contradicts the KB. The negation is derived as follows:

1. **Attempt to derive negation:**
1. \[ \text{opArg}_1 > 2, (\text{opArg}_1 + 6 \geq \text{opArg}_2) \rightarrow \text{opArg}_2 \geq 4, \text{opArg}_1 \leq 6 \]
   \[ \vdash (\text{opArg}_1 > \text{opArg}_2 \land \text{opArg}_2 \geq 6) \rightarrow \bot \]

2. \textit{Rule for implication}, prove consequent given antecedent:
   \[
   [\text{opArg}_1 > 2, (\text{opArg}_1 + 6 \geq \text{opArg}_2) \rightarrow \text{opArg}_2 \geq 4, \text{opArg}_1 \leq 6, \text{opArg}_1 > \text{opArg}_2, \text{opArg}_2 \geq 6] \vdash \bot
   \]

3. \textit{Law of Excluded Middle (Numerical Relations)} for \(\text{opArg}_1 > 2\):
   \[
   [\text{opArg}_1 > 2, (\text{opArg}_1 + 6 \geq \text{opArg}_2) \rightarrow \text{opArg}_2 \geq 4, \text{opArg}_1 \leq 6, \text{opArg}_1 > \text{opArg}_2, \text{opArg}_2 \geq 6, (\text{opArg}_1 \leq 2) \rightarrow \bot] \vdash \bot
   \]

4. \textit{Atomic Rule (Implied Goal)}, prove antecedent of the implication:
   \[
   [\text{opArg}_1 > 2, (\text{opArg}_1 + 6 \geq \text{opArg}_2) \rightarrow \text{opArg}_2 \geq 4, \text{opArg}_1 \leq 6, \text{opArg}_1 > \text{opArg}_2, \text{opArg}_2 \geq 6] \vdash \text{opArg}_1 \leq 2
   \]

5. \textit{Goal is inferred from Data}, (evaluation performed using Mathematica):
   \[
   [\text{opArg}_1 > 2, \text{opArg}_1 \leq 6, \text{opArg}_1 > \text{opArg}_2, \text{opArg}_2 \geq 6] \vdash \bot \rightarrow \text{opArg}_1 \leq 2
   \]
   Hence, proof of negation succeeds, i.e. \textbf{constraint contradicts the KB}.

This results in the second pre-condition being highlighted in red (unsatisfied constraint) in our KB tree, illustrated in \textit{Figure 5.3}.

The following OCL constraints test ability of our semantic evaluator to combine knowledge from different constraint types.

\begin{verbatim}
context Purchase::purchasePrice : Integer
  init : 0
  derive : if self.computePrice() >= 1000 then self.computePrice() else 0 endif

context Purchase::computePrice() : Integer
  post priceP1: let i:Integer = 10000 in result = i/10
  body priceB: 9999.9999999/10

context Car::driveTo(x:Real, y:Real) : Boolean
  pre driveP1: x > 0 and y > 0
  post: result = true -- irrelevant
  body: false -- since pre- fails

context Car::driveTo(a:Real, b:Real) : Boolean
  pre driveP2: not (a < 0) implies (b < 0)

context Purchase
  inv: let i:Integer = purchasePrice in
  i > 0 or i = 0
\end{verbatim}

The example combines pre-, post- and body conditions, initial and derived value expressions as well as an invariant. The screenshot of the resulting KB tree is illustrated in \textit{Figure 5.4}.

The tree displays the semantic order, in which constraints have been evaluated against the KB. Let us explain the colouring of tree nodes:
1. *init* constraint is the first to be added, and is hence highlighted in green (*satisfactory*), as it introduces new semantics to an empty KB:

\[ \text{purchasePrice} = 0 \]

2. *derive* constraint is the next to be added. By definition, our *init* constraint should satisfy *derive*. However, this is not the case, since by post-condition of computePrice(), its output is 1000, which satisfies the if-then-else condition in *derive*, and hence forces the latter to evaluate to the value of computePrice(), which is 1000, as already established:

\[ \text{purchasePrice} = 1000 \]

Since, by step 1, \( \text{purchasePrice} = 0 \), *derive* contradicts current specification and is therefore not added and highlighted in red in the KB tree.

Note that if-then-else expression is converted into the following logical form for our theorem prover:

\[ ((\text{computePrice}() \geq 1000) \rightarrow \text{purchasePrice} = 1000) \land \\
(\neg(\text{computePrice}() \geq 1000) \rightarrow \text{purchasePrice} = 0) \]

3. the *invariant* is added next. It tests the ability to refer to non-constant terms in *let* definition. It states that either purchasePrice is positive, or it is 0. Since we know already that the latter holds by step 1, the constraint is excessive and is therefore highlighted in yellow.

4. *pre-condition* “driveP1” is added. It simply gives lower bounds on arguments to *driveTo()*, and is therefore *satisfactory* (highlighted in green).

5. *pre-condition* “driveP2” is added. It specifies that \( \text{opArg1} \geq 0 \rightarrow \text{opArg2} < 0 \), which violates the lower bound specification for \( \text{opArg2} \) (second argument to *driveTo*) in step 4. Hence, the pre-condition is highlighted in red and represents a *contradictory* constraint.

6. post-condition “priceP1” trivially succeeds (highlighted in green), since it defines the output of *computePrice()* to be equal to 1000.

7. however, operation body “priceB” specifies a contradictory output to step 6, which tests the accuracy of numerical evaluation: the answer could be rounded up 1000 in the absence of sufficiently high precision. However, our evaluator succeeds and marks the constraint as *contradictory* (red).
8. Finally, post-condition and body operation for driveTo() are not evaluated, because pre-conditions to this operation are unsatisfiable (step 5). These constraints are referred to as skipped. Figure 5.3 displays an injection log for one such constraint.

This completes our testing of multiple constraint types.

We have thus provided a range of comprehensive examples, which test our semantic evaluation subsystem against a variety of targets, which we have discussed. The examples constitute only a fraction of the complete set of test cases used for semantic evaluation.
Chapter 6

Evaluation

In this chapter we evaluate our syntactic and semantic evaluation subsystems, which are part of OCL Evaluation Framework. Since this project has been about designing and implementing functionality in OCL, which goes beyond state-of-the-art software, the scope for quantitative evaluation of results is limited. Nevertheless, we attempt an evaluation of performance of our semantic subsystem, under significant simplifying assumptions.

We concentrate our discussion on the optimality of design choices made in OCL Evaluation Framework, since a wide range of alternative solutions has been considered at almost every major design step. Furthermore, we discuss the robustness of our semantic evaluator and its ability to accommodate further extensions and improvements.

6.1 Design Choices

Since one of our main objectives was to design a system, which provides complete support for OCL 2.0, there were two alternatives for the starting point of development: designing OCL Evaluation Framework from scratch or extending one of the existing open-source systems for OCL. We have decided to take the latter of the two for the following reasons:

- providing complete support for OCL 2.0 just on syntactic level would take significantly more time than the allowed frame for this project. Although, some parts of the grammar present exciting challenges in terms of syntactic evaluation, most of the implementation involves routine tasks (e.g. designing grammar rules in accordance with OCL abstract syntax, integrating a metamodel repository for managing UML models etc) and is not interesting from a non-commercial point.

- since the motivation behind the project was to provide functionality for OCL, which is beyond state of the art, why “re-invent the wheel”? We have shown that Dresden OCLParser presents a sophisticated, modular implementation of the lexer and parser for OCL 2.0 syntax, and luckily it has publicly available source code. In fact it perfectly suits our requirements, since its main purpose is to provide a base platform for building OCL software to other developers. Hence, we have decided to take Dresden OCLParser as a starting point, and have extended it to support OCL 2.0 specification almost completely. Also, it is implemented in Java, which facilities inter-platform OS compatibility.

Hence, we have chosen Dresden OCLParser as a starting point for development. Its other key advantage, is that the attribute evaluator is implemented in a single location,
which has enabled us to tightly integrate syntactic and semantic evaluation subsystems in OCL Evaluation Framework.

6.1.1 Constraint Solving Approaches to Semantic Evaluation

The obvious question that might arise in reader’s head is: why have we used Wolfram Mathematica to perform numerical evaluation, when it could be achieved using a constraint solver? Furthermore, why bother with cumbersome interfacing between Mathematica and Prolog, when one could simply plug in a constraint solver library into Prolog and perform all of the computation locally (i.e. without external delegation of routines)? Finally, why implement a theorem prover if one could employ a boolean constraint solver?

Constraint solving refers to a paradigm, which is based on the constraint satisfaction problem (CSP). For a user-defined situation, which is described by relations between decision variables, the CSP defines relations which should hold among specified those variables. Constraint solvers attempt to use various backtracking, branch-and-bound and other algorithms (based on techniques in AI, operations research, graph theory etc) in order to find satisfactory assignments to the decision variables (for a detailed overview of the field see [ConstSolving]).

There is a variety of widely used constraint solvers available. In our search for a more optimal solution than using external delegation to Mathematica from Prolog, we are interested in constraint solving libraries, which can be imported directly into Prolog and therefore used locally. SICStus Prolog is shipped with two constraint solvers:

- **clp(B) library** (implementation of methodology, proposed in [?]) can be used to reduce sets of constraints, specified as boolean expressions, only on boolean variables.
- **clp(Q,R) library** (see [SICStusCLP(QR)]) which can reduce sets of constraints on EITHER rationals (Q) OR reals (R). The library also implements some functions, such as abs(), min(), max() etc, which can be part of constraints. Note that one can either load the implementation for reals or for rationals, i.e. both cannot be loaded in the system at the same time!

The **clp(B)** library could in principle be used for manipulating logical connectives, however, it is incapable of representing numerical relations. Even assuming we had adopted our own approach, where numerical relations are represented as atomic facts, and dealt with simultaneously using entirely different routines, **clp(B)** is insufficient: it is unable to represent any expression other than a variable. Using clp(B) would require us to implement a set of conversion routines, which would map UML model element representations (our semantic form, e.g. attr(age,...)) to variables, manageable in clp(B). This leads to conversion overheads for every logical evaluation. Also, we cannot extend clp(B) with rules for managing numerical expressions (like we did with Law of Excluded Middle for Numerical Relations), which makes it practically unusable for anything other than logical expressions.

E.g. show ⊢ ((a > 100 → c < 5) → a > 10) → a > 1) (the aforementioned ((A → B) → A) → A) tautology) using clp(B). We must convert all non-logical connectives into variables; assume the following mapping:

(a > 100) ⇒ A, (c < 5) ⇒ B, (a > 10) ⇒ C, (a > 1) ⇒ D

Clearly, ⊢ ((A → B) → C) → D) cannot be reduced any further using clp(B), and our inability to extend it with matching rules for numerical relations (in our example, for variable mappings we have: A → C and C → D) make it an unusable alternative.

The main problem with clp(Q, R) library is that it can be used either only for integers (rationals Q) or only for reals (R). In general, the user can provide constraints combining
attributes and operations of both types, which means our evaluator should be able to deal with both. Using clp(Q,R) provides insufficient functionality, since one cannot combine constraints on both, Integer and Real domain, e.g. \{A + B > 4, A − B < 3.4\}.

In essence, the Reduce-Simplify sequence of routines in Mathematica, which we employ for numerical evaluation, comprises a constraint solving like procedure. Its advantages over pure stand-alone constraint solvers, is that it uses a variety of algorithms for reduction of constraints, and it can represent and manipulate a much wider range of numerical function calls. E.g. clp(Q,R) supports representing \(\sin(X)\) in expressions, but it cannot represent rounding of Real variables (e.g. \(\text{round}(X), \text{floor}(Y), \text{ceil}(Z)\)) and other numerical functions, which are not part of OCL at the moment, but will be added in future revisions. Hence, using clp(Q,R) is a poor approach in the context of extensibility: it cannot be modified to accommodate new numerical functions.

Also, clp(Q,R) cannot represent Strings. Even if we employ a hash function to convert strings to their unique numerical representations (e.g. hash\(F(\text{"peter"}) = 1234312421\)), clp(Q,R) cannot be used to represent operation calls for Strings. For \(\{\text{size}(x) = 4, x = \text{"peter"}\}\) it is critical that the constraint solver derives that size\(\text{x}\) refers to the size of string “peter”, which is 5. Therefore the set of constraints reduces to \(\bot\). Therefore we cannot use a hash function to represent strings, since semantics of string operations, such as \(\text{concat}(), \text{size}()\) etc cannot be represented. This is a problem of all pure constraint solvers in general.

Wolfram Mathematica, on the other hand, can represent numerical and string function calls in constraints. Furthermore, Reduce and Simplify operations encapsulate a multitude of powerful packages under a common name. These packages comprise multitudes of various algorithms, and Mathematica automatically determines the best sequence of reduction routines to apply to the given constraint set. This enables us to simply call the Reduce-Simplify sequence of routines, instead of initially trying to determine which sequence of algorithms is optimal. Figure 6.1 exemplifies evaluation of Strings, which will be executed in Mathematica

Figure 6.1: Screenshot of Mathematica Reduce-Simplify for Strings Example

by our KB evaluator for the following OCL specification. Suppose we have three attributes of type String in our UML class model, which are part of classifier “Container”: \(x, y\) and \(z\). Suppose the following OCL constraints are specified:

```plaintext
context Container
inv ySpec1 : x = "peter" and y = x.substring(1,4)
inv ySpec2 : y = "PETE"
```

Our KB evaluator will insert the first invariant into the KB successfully, since the latter is initially empty. But then for the second invariant, it will attempt to derive \([x = "peter", y = \text{substring}(x, 1,4)] \implies y = "PETE" \rightarrow \bot\), which our goal-directed theorem prover reduces to
Figure 6.1 illustrates how first the Data is reduced to \( \bot \) (contradictory constraints), which becomes the assumption for simplifying our goal. Since \( \bot \) implies anything, our Reduce-Simplify routine succeeds and hence, the theorem prover derives negation of \( y = "PETE" \) successfully, i.e. the second invariant introduces a contradiction into the KB and is therefore not added.

Due to the unique ability of Mathematica to combine different types in its constraint solving routines, it has been chosen for numerical and string evaluation in our semantic subsystem.

### 6.1.2 Alternative Computational Packages

We could have employed other computational packages, such as Maple and MATLAB, instead of Wolfram Mathematica. The idea of using Mathematica in the first place is due to my extensive past experience with this software.

The key advantage of Mathematica versus Maple and MathWorks MATLAB is that it is shipped with an official high-level interface, which allows to call Mathematica kernel from Java. This interface is called JLink, and it provides several simple and highly abstract routines (e.g. `kernel.eval(command)` for evaluating expressions) to perform computation in Mathematica. Thus our Prolog KB evaluator can simply call a proxy java class MathCall, which forwards the command to Mathematica kernel, receives response string and passes it back to the evaluator.

MathWorks provides interfaces for calling Java from MATLAB, but not the other way round. Third-party interfaces do exist, but they are not officially supported, and are version dependent. Maple does not provide an official Java interface at all.

This makes Mathematica the optimal alternative both, in terms of functionality, and in terms of well-defined support and accessibility for Java interactions.

### 6.2 Syntactic Subsystem

Attempting a quantitative evaluation of performance of our syntactic subsystem is not meaningful for three reasons. Firstly, it completes in under a second for test files, with 10-20 constraints, which we have found to be more than sufficient for definitions of medium-sized systems. Functionality and accessibility of OCL is still too limited to be used for commercial applications, and specifications of large and complex software. Secondly, there is lack of alternatives, which could potentially improve performance: implementations, which use other parser generators (not SableCC) are not publicly available and our attribute evaluator (\texttt{LAttrAstGenerator}, which is part of contextual evaluation) cannot be optimized significantly, because the AST generation procedure still has to conform the Abstract Syntax of OCL 2.0, as defined by the specification. Thirdly, the primary heuristic for evaluation of syntactic subsystem is the robustness of the system rather than performance. When performance involves overheads of only a couple of seconds, the limiting factors of the system are: how wide the range of supported OCL 2.0 expressions is, whether all possible errors are picked up by the syntactic evaluator etc.

The range of support for OCL 2.0 in OCL Evaluation Framework has been explored and illustrated in the previous chapter (see Section 5.3). The only known limitations of our support for OCL 2.0 are:

- no support for Enumerations. The main reasons behind this is that there are significant discrepancies between the way in which enumerations are converted into XMI
by UML modeling tools and the way in which they are accepted by metamodel-based repositories, which import XMI. Hence, compatibility issues between different implementations of XMI (discussed in Section 2.1.4) are the main barrier to implementation of Enumeration types in OCL, since they practically cannot be exported into our software via UML models (which are interchanged in XMI).

- **no support for OclMessages.**
- **ambiguous behavior of commonSuperType() operation.** This and the previous point have been explained in Section 3.1.

Considering state of the art in OCL, which has been presented in Section 2.5, OCL Evaluation Framework provides the most complete support for OCL 2.0 syntax, both at the parsing and at contextual evaluation levels. Extending Dresden OCLParser has enabled us to provide the widest range of OCL 2.0 functionality, since we have found their system to be the only practically usable OCL software with working syntactic evaluation.

### 6.3 Semantic Subsystem

In the context of OCL semantic evaluation there are three dimensions for assessment of our system:

- **robustness**, which refers to the ability of our system to correctly identify constraints which are contradictory and which are satisfactory.

- **range**, which defines the variety of OCL expressions, that our evaluator can reason with.

- **performance**, which refers to the time semantic evaluation takes to complete. Sophisticated semantic evaluation tends to take considerable time, and it is important to determine whether time overheads are acceptable for practical usage of OCL Evaluation Framework.

#### 6.3.1 Correctness and Range

Our case study has shown that our semantic subsystem is able to process all constraint types of OCL. It supports all of OCL constructs (let-in, if-then-else etc) and all built-in OCL types and pre-defined operations, which are supported by our syntactic subsystem. However, there are a few exceptions.

As already mentioned in specification of semantic subsystem, we are **not able to evaluate collection types**. A first-order logic (FOL) theorem prover would be required to represent collections, since quantifiers are needed to express elements of collections, e.g. expressing that there are some elements $x$ which are part of collection $S$, which satisfy certain property $p$. The problem with designing an automated FOL theorem prover is the **undecidability** of FOL:

- **Gödel's Completeness Theorem** states that any valid theorem in FOL can be proven, given unbounded resources. This means FOL is **semidecidable**, i.e. any algorithm is guaranteed to find a theorem, only if it is valid. It also means that FOL is **computably enumerable**, i.e. there exists an algorithm for enumerating all valid theorems, but it can run forever, if necessary.
• semidecidability property implies that invalid theorems in FOL cannot always be recognised, i.e. any semantic evaluation system is not guaranteed to determine invalid OCL expressions, which involve collection types, and can therefore produce no answer (not the wrong answer!).

• Numerical evaluation needs to be integrated into FOL theorem prover, since numerical relations can involve quantified variables (e.g. \( \forall x \) and \( \exists y \)), which means that numerical constraints cannot be reduced correctly without knowledge of context of variables involved.

• However, Gödel’s Incompleteness Theorems imply that FOL, extended with theory for natural numbers (not even considering Integers and Reals) contains a valid theorem, which cannot be proved by any algorithm and results in its non-termination. This means that our semantic evaluator can end up looping forever, while trying to prove a valid OCL constraint, which involves collections and numbers.

Nevertheless, restricted forms of FOL have been defined, which have resulted in sound (every theorem proved is valid) and complete (all valid theorems can be proved) automated proof systems. In Section 7.1 we discuss possible approaches to semantic evaluation of collection types.

However, for the range of supported OCL expressions, we have demonstrated on an extensive set of examples (see Section 5.3) that our semantic subsystem provides correct proofs for satisfiable constraints, correctly determines contradictory constraints and produces correct proofs for negations of these constraints. From a theoretical viewpoint, however, it is hard to demonstrate that our theorem prover is either sound or complete for the following reasons:

• the propositional goal-directed calculus has been proved to be sound and complete (GDProvers). We have provided an extension to the goal-directed calculus via Law of Excluded Middle for Relations. Purely in propositional logic, the rule does not affect either the semantics of Data or order of proof, as it introduces a new fact \( \text{newAtom} \rightarrow \bot \) into the Data, which involves an irrelevant atom. Hence our extended goal-directed calculus is sound and complete, but only within propositional logic.

• since we provide support of numerical relations via Mathematica, purely theoretical calculus cannot be defined anymore and we have to refer to implementation details in order to specify what our system does. This is because we cannot formally define operations performed by Reduce-Solve sequence of routines in Mathematica. Therefore we have to reason about soundness and completeness of our implementation in code, which leads into an even more obscure area of software correctness proof theory.

Hence, we have opted for the best feasible option which is to demonstrate “intuitively” that it works on a wide range of examples.

6.3.2 Performance

NP (Non-deterministic Polynomial time) defines a complexity class of problems, which can be solved in polynomial time (e.g. time complexity of \( O(n^k) \) is of polynomial order \( k \), where \( k \) is some problem-dependent constant), by a non-deterministic Turing machine. It contains P, which defines a class of problems, which can be solved in polynomial time but by a deterministic Turing machine. **NP-complete** (NPC) is a subset of NP, for each problem \( p \) in NP-complete, *every* problem in NP can be reduced to \( p \) in polynomial time. Moreover, this implies that each NP-complete problem can be reduced to any other NP-complete problem.
in polynomial time \((NPC \subset NP)\). NP-complete problems are an interesting subject of study, because no polynomial order algorithms have yet been found to solve any of these problems. But if at least one is found, all NP problems can be solved in polynomial time (since they reduce to NPC in polynomial time) and, in fact, \(P = NP\) (since no polynomial NPC algorithm has been found, it is widely believed that \(P \subset NP\)).

Our theorem prover is based on propositional logic, and essentially attempts to solve the Boolean Satisfiability (SAT) problem, which is a general task of determining whether a logical expression (consisting only of atoms and boolean connectives) evaluates to TRUE or FALSE. SAT is known to be decidable but NP-complete, which implies that the time complexity of our prover in general is exponential at best. Still, it is important to measure time overheads for practical examples, in order to see whether semantic evaluation is fast enough to be used in practical scenarios.

In order to measure increases in time overheads, with increasing semantic complexity of OCL constraint set, we need to define two notions:

- what is meant by semantic complexity in the context of OCL constraints
- how can we increase semantic complexity in systematic way, so that results can be comparable across experiments

Consider \(\vdash ((A \to B) \to A) \to A\). We have shown that our goal-directed prover takes 11 or so steps to derive this tautology. On the other hand, \(\vdash A\) involves a single goal-directed step. We can say that OCL constraints, which reduce to these logical forms, have different semantic complexity, and the first example is much more complex than the second.

To measure time overheads in a comparative way, we cannot simply take a constraint and keep introducing arbitrary additional semantic complexity for measurement. This would mean that time overhead resulting from going from constraint set \(A\) to \(B\) (where \(B\) is more semantically complex than \(A\)), is not comparable to time overhead of going from \(B\) to \(C\) (where \(C\) is more semantically complex than \(B\), but to an arbitrary extent).

We propose the following general way for comparative analysis of time overheads: for a formula \(\vdash A :a: B :o: C\), where :a: and :o: are some logical connectives, a chain of comparative increasing complexity for this formula can be defined as:

\[
P \vdash A :a: B :o: C \implies P \vdash (A_1 :a: B_1 :o: C_1) :a: B :o: C \implies P \vdash (A_1 :a: B_1 :o: C_1) :a: (A_2 :a: B_2 :o: C_2) :o: C \implies P \vdash (A_1 :a: B_1 :o: C_1) :a: (A_2 :a: B_2 :o: C_2) :o: (A_3 :a: B_3 :o: C_3) \implies \ldots
\]

Hence we will construct two examples in OCL:

- **Tautology.** OCL invariant, which specifies \(\vdash ((A \to B) \to A) \to A\) tautology. We will extend it in accordance with the general chain and measure overall execution time of semantic evaluation at each step.

- **De Morgan’s Theorem.** OCL invariant, which specifies De Morgan’s Theorem: \(\neg(p \land q) \vdash \neg p \lor \neg q\). We convert this to the tautology: \(\neg(p \land q) \to (\neg p \lor \neg q)\). The goal-directed proof involves 9 steps, including the restart rule.

It does not matter which type of constraints specify these rules, since expressions will be converted into the same semantic form. Note that our theorem prover will perform two computations at each step of the complexity chain: attempt to derive negation of the goal first (which will fail, since examples are tautologies), attempt to prove the goal (which will succeed; hence constraints will always be excessive).
For each step in the chain, we measure two attributes: overall execution time of semantic evaluation (in ms) and number of proof steps (rules applied), performed by our goal-directed prover. All execution times are averages over five trials, which have been taken to avoid outliers. Results for the Tautology example are illustrated in Figure 6.2.

It shows graphs of attributes, recorded at each step of the complexity chain. $a^*$ denotes that a itself is not an atom, but a recursive definition of the same overall expression, e.g. $((a \rightarrow b^*) \rightarrow a) \rightarrow a$ can be written in full as:

$((a \rightarrow [(a \rightarrow b) \rightarrow a]) \rightarrow a) \rightarrow a$.

From the graph we can observe that the execution time more than quadruples (328 ms to 1248 ms) as each atom $a$ becomes a recursive definition $a^*$, while the number of proof steps only doubles (12 to 22). This is due to the fact that the first proof step, translation to RFC form, takes significantly longer than it would for a non-recursive form: $a \rightarrow b$ is already in RFC form, but $a \rightarrow b^*$ is not, because $b$ must be atomic. Hence, because $b^*$ itself represents the tautology, each $b^*$ in the expression needs to be translated recursively. In our example, we have inserted three recursive forms $a^*$, which practically triples the time it takes to convert to RFC. Indeed, our proposition is confirmed by the low time increase from 1248 to 1408 ms as the only difference between the last two states of the complexity chain is a single addition of $b^*$.

The assumption that each proof step takes the same time to execute on average gives a decent fit (proof steps curve has a similar slope to the execution time curve) for the first two states of the complexity chain, where the only difference is an addition of one extra recursive form for $b^*$. But our hypothesis about translation to RFC is indeed confirmed, as the fit collapses, when we insert three recursive forms into the next state in the chain. Overall, the upper bound of 1404 ms (1.4s) is an indication of good performance, given that in most practical cases logical specifications for a software system do not get as complex, due to the fact that their semantics is extremely unclear in English.

Figure 6.3 shows a similar plot for our second example, De Morgan’s Theorem. For the first three states of the complexity chain, the slopes of two curves are approximately equal, which means that the time for each proof step is on average the same, i.e. increases in execution time are due to more rules being applied by our goal-directed prover. Time complexity increases dramatically (772 to 3198 ms) as recursive forms for $a^*$ are added, but drops significantly (to 1857 ms) when $a^*$ forms are replaced with $b^*$. This seems to indicate anomalous behaviour, given that $a$ and $b$ are symmetric with respect to De
Morgan's Theorem. Discrepancies are due to the fact that in our OCL setup we cannot define free variables, and hence are force to initialise them: \( a \) evaluates to \( \bot \), while \( b \) evaluates to \( \top \). These introduce different complexity to the proof (e.g. proving \( \top \) always succeeds directly, while proving \( \bot \) can in general take arbitrary time). Also, recall that disjunctions get converted to implications in goal-directed calculus, hence proving the first disjunct is asymmetric to proving the second disjunct via goal-directed methods, because disjuncts become parts of an implication, which is asymmetric.

The relationship between two curves seems to break for the last state of the complexity chain, where both \( a^* \) and \( b^* \) recursive forms are present: an increase of proof steps from 23 to 58 does not justify more than a 7 times increase in time execution to 13.5 seconds! The only viable explanation is that translation to RFC form takes much longer than for the previous state. This could indeed be true, since just negations result in an extra recursion step for our translation routine \( (\neg a \Rightarrow (a \rightarrow \bot)) \), and disjunctions can take a significant number of translation steps. The upper bound of 13.5 seconds would be on the border of being unacceptable for a single constraint. However, similar to the Tautology example, it is hard to come up with practical scenarios, where such an expression could be specified as an actual constraint on a software system.

Hence, we have established that for practical purposes, the time complexity of our semantic evaluation subsystem is satisfactory. Also, note that in our examples we have not used numerical relations. This has been done to illustrate that time overheads, associated with communication with Mathematica are truly minimal, since execution of semantic evaluation on constraints with numerical relations takes similar amounts of time. Hence the only significant performance optimisations can only result by improving routines of our goal-directed prover in Prolog.

### 6.4 Graphical User Interface

This project has been centered on providing functionality for OCL. Our main emphasis in the context of user front-end has been on providing a simple interface to functionality in OCL. Of course, our GUI lacks features of most commercial software:

- it lacks a powerful constraint management system, which would allow one to create
projects, containing several constraint files, which could be loaded for testing selectively by the user.

- the readability of syntactic output, such as error messages and OCL constraints, could be further improved with syntax highlighting and on-the-fly evaluation (as opposed to manually executing syntactic evaluation) of constraints.

- loaded UML class models could be displayed graphically and manipulated directly within OCL Evaluation Framework.

- output of semantic proofs could be structured as a tree, each node corresponding to each rule applied in the proof. This would enable user to expand and collapse tree nodes and thus make constraint injection logs much more readable.

However, the primary purpose of our GUI is demonstration of concept and functionality, rather than a complete commercial implementation. All of the features above are purely cosmetic and could be implemented without any modifications to syntactic and semantic subsystems. This is because our GUI is implemented as a stand-alone class and is simply one view on the system, as specified by the Model-View-Controller design pattern.

### 6.5 Extensibility of Semantic Subsystem

We have illustrated in Figure 4.2 that our semantic evaluation subsystem is highly modular, i.e. it can be extended to accommodate additional operations, and types (non-collection only) with minimal modifications. For example, inserting a new operation `string.take(5)`, which returns the first 5 characters as a string, in OCL can be done as follows:

1. modifying `addStringOpExpr(...)` method in ConstraintPacker with a new clause, which specifies the semantic form for the operation, e.g. “takeStr(string, 5)” could be an appropriate representation for our example. Recall, that semantic form represents a fact (unary predicate) in our Prolog KB Evaluator.

2. inserting a new clause for `opToMath/3` predicate (`mathcall.pl`), which specifies the corresponding Mathematica function. For our example, the corresponding Mathematica function is “StringTake[string, 5]”. Hence we simply a clause of `opToMath/3` which converts the semantic form into this string for Mathematica.

3. inserting an additional printing clause (in `print.pl`), which specifies how to print this predicate in a readable form for constraint injection logs (e.g. `string.take(5)`, same way as it is entered in OCL). This is optional, as the semantic form will be printed as-is by default.

We have thus shown that only three steps are required to implement support for a new OCL operation in our semantic subsystem. This is a key advantage of our system, since new versions of OCL may introduce significant revisions of syntax, built-in types and operations.
Chapter 7

Conclusions

We have shown that it is possible to design a complete parser for OCL 2.0 syntax, which can perform generation of the AST, in accordance with the Abstract Syntax of OCL 2.0, in one traversal. We have also shown that semantic evaluation in OCL is feasible, and can be designed using a goal-directed theorem prover (in Prolog).

It has been demonstrated that a propositional logic prover can be extended to support numerical relations, and it does not require first-order logic (FOL). It has been tested on practical examples, and is sound for those examples (although the proof for general case has not been attempted). We have also shown that tight three-way integration of Java-Prolog-Mathematica is practically feasible and does not result in significant time overheads in performance of semantic evaluation subsystem. It has been illustrated that our semantic evaluation handles a wide range of OCL expressions, produces correct output for these expressions. Although we have shown generic semantic evaluation in OCL to be NP-complete, it results in acceptable time overheads (up to 13.5 seconds) for most practical applications.

We have determined that semantic evaluation for OCL collection types is, in general case, undecidable and that the set of decidable OCL expressions has to be defined before generic semantic evaluation can be attempted.

7.1 Future Work

Implementing semantic evaluation for collection types remains an open challenge, which could be attempted in the future. Restrictive axioms have been defined for FOL, which has resulted in automated provers, which are sound and complete. Attempting comprehensive semantic evaluation of collection types in OCL requires one to determine a suitable restriction of FOL, which in turn determines which theorem provers can potentially be used for the task. As we have discussed, one existing approach to semantic evaluation, HOL-OCL, attempts to use an interactive theorem prover, Isabelle, which often cannot provide automated proofs and requires proof hints from the user. For an accessible software system this is not a viable option. A reasonable approach is to attempt to define a set of OCL expressions, which are decidable and insert manual proof routines and theorems for OCL into Isabelle, so that evaluation of these expressions can be fully automated.

In addition, there is much more scope for semantic evaluation in OCL. We present other dimensions are useful and should be investigated:

- **Simulation of Constraints.** A complete evaluation requires an object instance snapshot of the system. A useful type of semantic evaluation is an execution of user-
defined constraints on a snapshot of the system, which conforms to the underlying UML class model. The main barrier to implementing such simulations is the absence of means of exchanging such snapshots (reasons for this are discussed in the XMI subsection of Section 2.1 on metamodels).

- **Runtime Evaluation of Constraints.** Invariants, pre and post conditions evaluate to type Boolean. It is useful to evaluate these OCL constraints against a running software implementation of the system, specified by the underlying UML metamodel. Implementation of this type of semantic analysis defines several requirements on the software in that the latter has to follow a very model-driven architecture (e.g. MDA or MOF, proposed by OMG and discussed above), which uses UML throughout development. A key risk in this case is the absence of a programming language which is in complete synchronization with the integrated UML/OCL metamodel. However, this type of semantic evaluation could be implemented by injecting constraints into source code, which are executed by compiled software itself as opposed to stand-alone OCL software.

- **Automated Generation of Test Cases.** Constraints can be evaluated to determine the test space, i.e. the set of all possible test cases for the software, given its OCL specification. This kind of semantic evaluation can lead to automated testing of software systems.

(Currently available OCL functionality is discussed in [OCLFunction](#)).

In the context of syntactic evaluation, support OclMessages and Enumerations can be implemented in the future, although currently there is a limit to how complete their support can be, due to shortcomings of the OCL 2.0 specification itself.

As has been mentioned in Evaluation chapter, the front-end could be improved cosmetically, in particular in terms of error messages and semantic evaluation output. This is critical if the system is to be targeted for commercial applications.

Finally, from a theoretical viewpoint, one is interested in investigating whether our extended theorem prover actually presents a sound and complete system for propositional logic and numerical relations. If the Reduce-Simplify procedure in Mathematica could be formalized, a proof of our extended goal-directed calculus can be attempted. Alternatively, one could attempt to define reduction procedures directly in Prolog, although that would require a significant effort to equal the level of Mathematica. Support of Strings and run-time representation of operation calls (e.g. abs(3), min(3,5) etc), which are so elegantly handled by Mathematica, might prove to be quite complicated to implement in Prolog.
Bibliography

[ConstSolving] Constraint Programming. F. Rossi, P. Van Beek, T. Walsh.
   Handbook of Knowledge Representation, F. van Harmelen et. al., 2008.

   Technical University of Dresden, Software Engineering Group.
   http://dresden-ocl.sourceforge.net/

   Technical University of Dresden, Software Engineering Group.
   September 2005.

   ISBN 0137263651. Ellis Horwood Ltd Publisher.
   March 1998.

[GDProvers] A Family of Goal Directed Theorem Provers Based on Conjunction and Implication.
   D. M. Gabbay and F. Kriwaczek.
   Imperial College London, Department of Computing. Journal of Automated Reasoning

   http://www.seop.leeds.ac.uk/entries/logic-higher-order/
   December 2007.

[HOLOCL] HOL-OCL: A Formal Proof Environment for UML/OCL.
   A. D. Brucker, SAP Research Germany.
   B. Wolff, Information Security Switzerland.
   2008.

[Isabelle] Isabelle Theorem Prover.
   L. Paulson, University of Cambridge, Department of Computing. T. Nipkow, Technical
   University of Munich.
   http://www.cl.cam.ac.uk/research/hvg/Isabelle/index.html


Dresden University of Technology, Department of Computer Science.

Polytechnic University of Catalonia, Department of LSI.

http://www.gentleware.com/products.html

[SableCCIntro] *SableCC Compiler Generator*. The SableCC Project.
http://sablecc.org/

[SICStusCLP(B)] *Methodology and Implementation of a CLP System*. J. Jaffar, S. Michaylov.

[SICStusCLP(QR)] *OFAI clp(q,r) Manual*. C. Holzbaur.
Austrian Research Institute for AI, Vienna, 1995.

[SICStusDocs] *SICStus Prolog*. Swedish Institute of Computer Science (SICS).
http://www.sics.se/isl/sicstuswww/site/documentation.html


University of Bergen, Department of Informatics.