Design and Implementation of a UML/OCL Compiler

Faridah Liduan

Master Thesis
INF/SCR-04/31

June 2004

Institute of Information and Computing Sciences
Utrecht University

Abstract

The Dutch Tax and Customs Administration (DTCA) conducts a research program called POWER (Program for an Ontology based Working Environment for Rules and regulations). The POWER program aims to support the whole chain of processes from drafting the legislation to implementing the law enforcement. Central in the POWER research program is the POWER method. This method is based on the conceptual modeling of legislation and regulations into formal legal specifications, and needs to produce a representation of legislation that computers can reason with.

Within the POWER project, the conceptual models are created using the industry standard Unified Modeling Language (UML). Part of the UML is the Object Constraint Language (OCL), which is used to express constraints that apply to the model.

This thesis project focuses on verifying the conceptual models to produce well- typed models, and generating knowledge based components from the well-typed models.

Type checking makes sure that the OCL expressions comply to the UML model which they constrain. During the type checking process, the types of the OCL expressions are computed. The type information acquired in this way is used in the code generation process and for validation purposes.

In this research project, the target of the code generation process is not directly executable. Instead we have chosen to use RBML as an intermediate language. RBML contains all the properties for a knowledge based system in the context of the POWER project. The generated RBML document can be mapped to the target languages that satisfy all the properties. In this way, we can generate code for any suitable target language.

An advantage of the RBML is that the VALENS verification tool can be used for verifying the RBML document. VALENS is a knowledge verification tool developed by a third party in collaboration with the POWER project.

Acknowledgements

This thesis project is mainly done in the Dutch Tax Office (Belastingdienst), Utrecht. Therefore, I would like to thank my supervisor Prof. Dr. Tom M. van Engers for giving me a chance to work on this project and for his continuous feedbacks and comments. He also creates a motivating and supportive atmosphere in the project.

I would like to thank my supervisor at the university, Dr. Jurriaan Hage. His critical questions and suggestions have improved this work extremely. While formalizing the OCL type system, I had several discussions with Bastiaan Heeren. During the discussions with him, we often came up with new ideas and solutions. I would like to thank him for his comments, corrections, and for being very helpful.

The code generation part of this thesis project has been done in LibRT B.V. For this, I would like to thank Rik Gerrits and Silvie Spreeuwenberg. They have been very helpful in providing the information I needed and in giving explanation about RBML. I also had many fruitful discussions with them.

What also makes this project stimulating is to work together with the other students, Kamal Sayah, Niels Egberts, and Wilco Niessen. I would like to thank them for their help and cooperation, and for many cheerful moments in the project.

The first person I would like to thank in general is Prof. Dr. Doaitse Swierstra. He provides a great help since the beginning I joined the Master Program Software Technology, and always welcomes questions and discussions. I also would like to thank my mentor Dr. Wishnu Prasetya for his useful suggestions and for his help during my studies in the Utrecht University.

Next, I would like to thank Desy for being a good friend, for being the one to share with, and for making me not missing home too much.

Last but important, I would like to thank my parents for their love and care, and for giving me an opportunity to be an independent individual. Thanks to my sisters for their love and for a great time we have spent together. And thanks to Johnson for his patience and continuous support throughout the years.

Contents

Chapter 1

Introduction

The Dutch Tax and Customs Administration (DTCA) has initiated a systematic translation of legislation by conducting a research program called POWER (Program for an Ontology based Working Environment for Rules and regulations). A detailed description of this research program can be found in the paper by Tom M. van Engers et al. [EK98].

Central in the POWER research program is the POWER method, which aims to support the whole chain of processes from legislation drafting to implementing law enforcement. This method is based on conceptual modeling of legislation and regulations into formal legal specifications, which wants to produce a representation of legislation that computers can reason with.

The POWER process, as described in the paper by Tom M. van Engers et al. [vGB+01], consists of five sub processes:

1. The translation of legislation and regulations to conceptual models, including completion of the models by expert knowledge elicitation.
2. The refactoring of conceptual models into coherent conceptual models.
3. The verification of conceptual models, including the detection of incompleteness and identification of missing legislation and regulations.
4. Generating knowledge-based components for application frameworks, thus creating knowledge based systems that can be used for implementing law enforcement.
5. Testing and validating knowledge components, including the involvement of experts for certification of the knowledge components.

This research project is aimed to design and implement tools that can be used to execute the POWER process. We will focus on verifying the conceptual models, as in point 3, to produce well-typed models, and generating

knowledge based components from the well-typed models. In the context of POWER project, a knowledge based component is defined as *a rule based system that supports interfaces to an application framework to perform automated knowledge intensive tasks* [vGB+01].

The translation of legislation and regulations into conceptual models produces UML models. Additional constraints for the UML models are written in the form of OCL constraints as part of the models. From now on, we will use UML/OCL to refer to the UML models with OCL constraints.

All classes, types, interfaces, and datatypes defined in the UML model define types within OCL. These types are called *model types*, which together with the predefined OCL types form the valid OCL types. In order to have valid OCL expressions, we need to check that each OCL expression has a type: either a model type or one of the predefined OCL types. Type checking is necessary to make sure that the OCL constraints defined comply to UML model to which they belong. Also, type information is needed for generating knowledge based components. This infomation is computed during the type checking process.

The following examples show OCL expressions that have type errors.

```
let discount:Boolean = true
in discount * 0.1
```

In this Let expression, we declare a variable `discount` of type `Boolean` with an initial value `true`. Then we use the declared variable in the body of the Let expression by multiplying it with the numeric value 0.1. This expression is syntactically correct, but it is not well-typed because we multiply a boolean value and a numeric value.

```
if isActive=true
then 1000
else 'unknown'
endif
```

In this example, we assume the attribute `isActive` is in the current context and of type `Boolean`. If the value `isActive` is `true` then an integer value 1000 is returned and otherwise the string value `'unknown'`. This expression is a syntactically correct expression too, but it is not well-typed because the `then` and `else` parts have different types.

There are still many possible errors that can be made in writing OCL expressions. The type checking is intended to detect the type errors, and report with appropriate error messages to help users identify and correct the errors.

Based on the conceptual models, knowledge based components for the application frameworks are created.

There already exists a system which generates knowledge based components from UML/OCL within the POWER project, called FORCE. FORCE is written in Aion, a development tool for component based expert systems from Computer Associates.

FORCE only performs syntactic checks on the input OCL expressions. No type checking is done. Therefore, it is possible that the generated knowledge based components by FORCE are not well-typed and may result in run-time exception later on.

Also, since the POWER method aims to preserve a maximum independence of implementation issues, we would like to map our UML/OCL models to a general system, from which we can generate code for any programming language. This system will serve as an intermediate system between UML/OCL and the target languages.

The general system should contain all the concepts for the knowledge based system in the context of the POWER project [vGB+01], namely, it should be

- Object Oriented (OO)

  This requirement is a result of UML/OCL being object-oriented.

- Rule-based

  A rule based system consists of a set of IF-THEN rules used to derive new facts from given facts. These rules are called *inference rules*. Inference rules are executed by an inference engine.

  An inference engine provides an algorithm for executing potentially unordered statements by finding connections between those statements in order to resolve unknown values by means of known values. It precludes restructuring the knowledge into a procedural model that has to be evaluated in a particular order.

Since this conceptual system does not refer to a specific industry standard product, the choice for the system to be used has been determined in this research project.

We choose to use the Rule Based Markup Language (RBML) for the general system. RBML is a domain specific language for a rule based representation [Libb]. It is developed by the LibRT, an associate partner of the POWER project.

The reasons that RBML was chosen as our target language are:

- RBML is *object oriented* and *rule based*.
- The generated output can be validated and verified using VALENS verification tool. VALENS is a knowledge verification tool developed by a third party in collaboration with the POWER project.

  For more information about VALENS, see the website of the LibRT [Liba].

In the code generation process, we generate the RBML document from the input OCL expressions. The generated RBML document can be mapped to any target language that satisfies the properties for the knowledge based components in the context of the POWER project.

1.1 The Compiler Architecture

We now present the architecture of our UML/OCL Compiler. We will divide the processes into several parts, each performing different responsibilities.

The UML/OCL compiler consists three main parts, namely:

- Parser

  The parser does the syntactic analysis of the input OCL expressions. It transforms OCL expressions into an OCL abstract syntax tree (AST) according to the OCL 2.0 abstract grammar [OMG03]. If there are syntax errors, they will be reported in terms of the source OCL expressions.

- Type Checker

  If the parsing of OCL expressions succeeds, the compilation process continues with the type checking. The type checker reads the abstract syntax tree produced by the parser and performs the type checking. The type errors detected will be printed in terms of the source OCL expressions. After the type checking process, the type information is available at the nodes of the tree.

- Code Generator

  If there are no type errors found during the type checking, the compilation process will continue with the code generation. The code generator generates a RBML document from the OCL abstract syntax tree that has been enriched with the type information. A RBML document is an XML document that conforms to the RBML schema. RBML represents rules in the object oriented context. It is developed by LibRT, an associate partner of the POWER project.

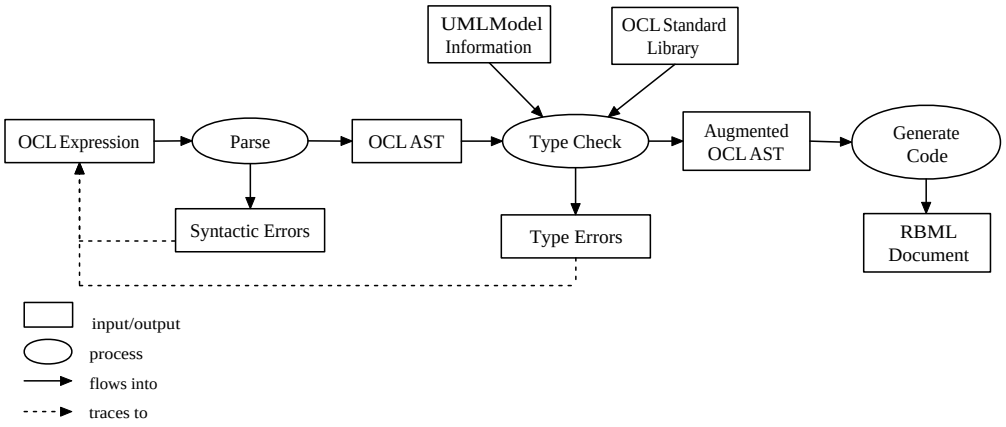


Figure 1.1: *Architecture of the UML/OCL compiler*

Figure 1.1 shows the architecture of our design.

In the next chapters, we will explain each phase in detail.

1.2 Related Work

There are number of researches on OCL with varying purposes and approaches.

D. Moude Foko in his thesis [Fok02] has designed a first version of a compiler implementation for OCL, which includes parsing, type checking, and code generation. However, the code generation process has not been completed yet. The compiler is implemented using Haskell and Attribute Grammar (AG) system. His research has been done in the context of POWER project and our research can be seen as the continuation of his work.

The Dresden OCL Toolkit is a modular toolkit for OCL support [oT]. It is based on an OCL compiler developed by Frank Finger at the Dresden University of Technology. Detailed documentation of the compiler can be found in the thesis of Frank Finger [Fin00]. The compiler consists of several modules, namely a parser, semantic analysis, normalization, and code generation. The Java source code for the parser is generated out of a grammar description using the SableCC parser generator. The semantic analysis does simple consistency checks and type checking. Normalization facilitates the code generation processs by simplifying the AST produced by the parser. The Java code for the OCL expressions is generated by the code generator.

USE (UML-based Specification Environment) is a system for the specification of information systems [oB]. A USE specification contains a textual description of a model using features found in UML class diagrams. Expressions written in the Object Constraint Language (OCL) are used to specify additional integrity constraints on the model. The USE tools present an approach for the validation of UML models and OCL constraints, based on animation. It is described by Richters in his thesis [Ric02].

This thesis is organized as follows, Chapter 2 gives an overview of the UML/OCL. By using an example of the class diagram, we show the various kinds of OCL expressions and their use in the context of a UML model.

In Chapter 3 we explain how the parsers are implemented using the Utrecht Parser Combinators (UPC). Chapter 4 introduces the formalization of the OCL type system. We formalize the type rules for each construct of the OCL, show some examples of the type derivation, and then explain the type checking process as a whole. In Chapter 5 we explain our approaches in the translation from OCL to RBML and present the translation schemes for the code generation process. Finally, in Chapter 6 we conclude this work.

Chapter 2

UML/OCL

This chapter gives an overview of UML/OCL models. In Section 2.1 we give an overview of the UML model. By using an example of the UML class diagram, and we explain the common terms which are used to refer to a certain element of the model. In Section 2.2 we give an overview of the OCL. By first listing several kinds of context declarations, we then explain the *classifier* context declaration. We show how the various constructs of OCL can be written in this context declaration to constrain model of a domain. OCL types and their predefined operations are discussed along the way, while describing the expressions. Finally, we show the OCL type hierarchy and explain the remainder of the context declarations.

2.1 Unified Modeling Language

The Unified Modeling Language (UML) is a language and notation for specification, construction, visualization, and documentation of models of software systems.

The UML notation is largely based on diagrams. However, for certain aspects of a design, diagrams often do not provide the level of conciseness and expressiveness that a textual language can offer. Textual annotations are frequently used to add details to a design. For this purpose, the Object Constraint Language (OCL) provides a framework for specifying constraints on a model in a formal way.

OCL is a pure specification language. It is side-effect free. When an OCL expression is evaluated, it simply returns a value. It cannot change anything in the model. This means that the state of the system will never change because of the evaluation of an OCL expression.

Figure 2.1 shows an example of a UML class diagram. This example will be used to illustrate OCL in the next sections.

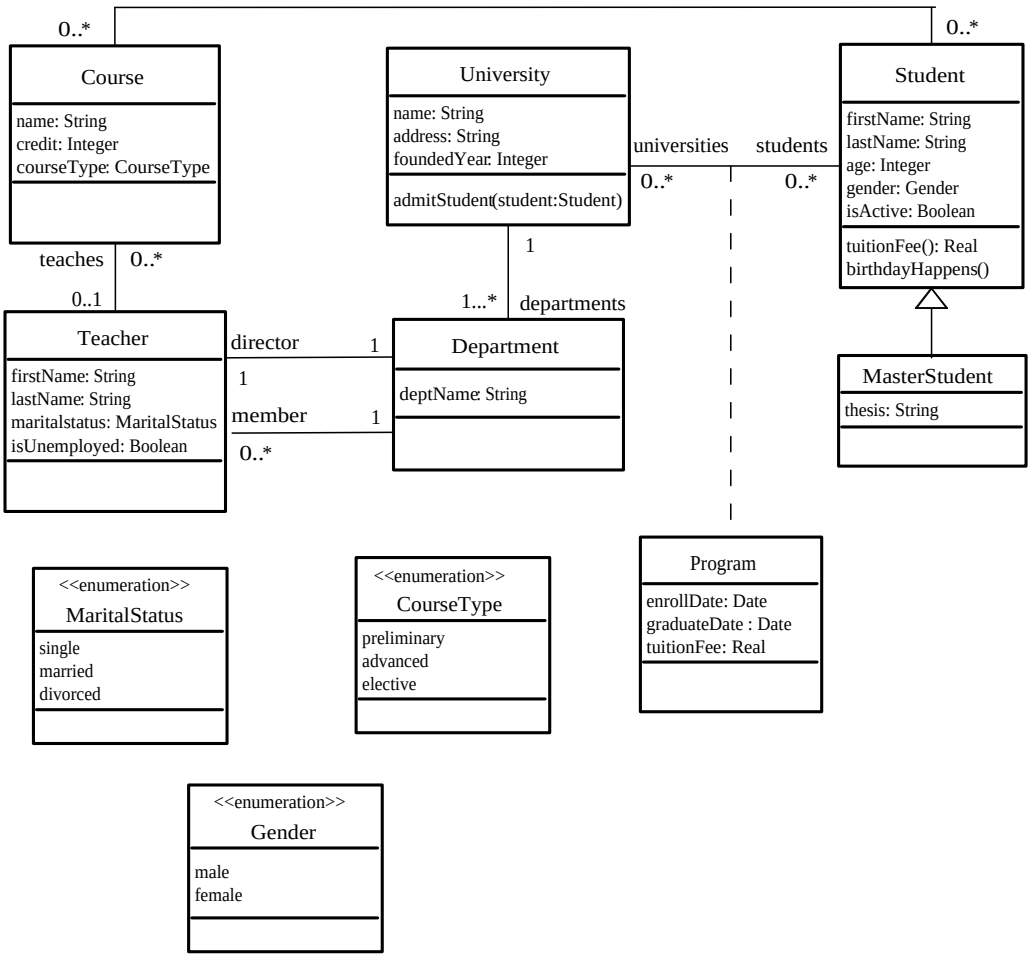


Figure 2.1: *Class diagram example*

We use the example class diagram to model the university system. The class `University` has two associations: to the class `Department` and to the class `Student`. In particular, these associations are association ends. An association end can have a name. If the name is empty, we use the same name as the name of the object at the association end, but starting with the lower case letter. For example, we can refer to the association between the classes `Teacher` and `Department` as `department`. Each association end has a multiplicity, which can be one of the following: "0..1", "*", "1..*",or "1". A multiplicity represents the maximum number of the associated classes at the opposite end of an assocation. When it is not specified, the default multiplicity is 1.

The class `University` has three attributes: `name`, `address`, `foundedYear` and one operation: `admitStudent`. Each attribute has a type. An operation on the other hand can have parameters (within the brackets) and a return type (after the brackets, preceeded by a colon). The operation `admitStudent` takes one parameter and returns no type, while the operation `tuitionFee` of the class `Student` takes no parameter and returns a real value.

If the association of two classes results in the needs for common attributes, it ends up with creating an association class. The only association class in this model is `program`. The classes `University` and `Student` can navigate to this association class to retrieve the class `Program`. The class `Program` owns the common attributes between the two classes.

The class `MasterStudent` is a subclass of `Student`, which means that it inherits all the properties defined for the class `Student`. A property is an attribute, operation, or association end of a class. The class `Student` is a generalization of the class `MasterStudent`.

An enumeration defines a new type by listing all its possible values. This model defines three enumerations: `MaritalStatus`, `CourseType`, and `Gender`. We can use an enumeration datatype to define the type of attribute or operation. For example, attribute `maritalstatus` of the class `Teacher` has an enumeration type `MaritalStatus`.

In OCL, the types, classes, interfaces, associations, and datatypes from UML model are referred to as *classifiers*. Each classifier defined within UML model represents a distinct OCL type. Classifier is a base type for OCL.

For more information on UML, see the book of Fowler et al. [BJR00].

2.2 Object Constraint Language

The Object Constraint Language (OCL) is a formal language used to describe expressions in the context of a UML model. These expressions typically specify constraints that must hold for the system being modeled. An

OCL constraint is a valid OCL expression of type *Boolean*. It evaluates to *true* if the restriction holds.

Although OCL is initially used for describing constraints, OCL 2.0 specifies the Object Constraint Languages as a general object query language that can be used wherever expressions over UML models are required [OMG03].

OCL is a pure specification language, therefore an OCL expression is guaranteed to be sife effect free. When an OCL expression is evaluated, it simply returns a value. It cannot change anything in the model [OMG03].

An OCL expression is always written in some syntactical contexts. The context of an OCL expression within a UML model can be specified through a so-called context declaration. A context declaration is written at the beginning of an OCL expression. When we write a context declaration, we introduce a new scope for the OCL expressions written after it.

There are three kinds of OCL context declarations, namely: *classifier* context, *operation* context, and *attribute or assocation* context.

We first discuss the *classifier* context since most examples in the next sections are written under this context declaration. We will postpone the discussion of other context declarations in the last section.

A *classifier* context declaration is used for the expressions that can be coupled to classifiers. The constraints that can be defined after this context declaration are *invariant* and *definition*.

The syntax of the classifier context declaration is as follows:

```
context Typename
inv: -- some expression to constrain the classifier
def: -- a  helper attribute /operation definition
```

Invariants

An *invariant* is a constraint that states a condition that must always be met by all instances of the classifier. An invariant is described by using an expression that evaluates to true if the invariant is met [CW02].

By using the UML class diagram example in Section 2.1, we can constrain the attribute `age` of the class `Student` by writing the following invariant:

```
context Student
inv: self.age >= 18
```

This invariant states that every student has age at least 18. The variable `self` is used to refer to the instance of the context to which the expression is attached. In this case, `self` refers to the instance of the class `Student`.

Definitions

A *definition* constraint enables us to define a helper variable or operation which can be reused over multiple OCL expressions.

```
context Teacher
def: courses:Integer = self.teaches -> size()
def: teachCourse(c:String):Boolean
       = self.teaches->exists(course=c)
```

The first constraint defines the attribute `courses`, which is the number of courses taught by a teacher. The second constraint defines the operation `teachCourse`. Given the name of a course, the operation determines whether a particular teacher teaches that course.

The variables and operations defined in the `def` constraint are known in the same context as any property of the classifier. Therefore, their names may not conflict with the names of the normal properties of the classifier. The variables and operations are used in an OCL expression in exactly the same way as normal attributes or operations are used.

Although many `def` constraints can be written in one context declaration, no mutual recursion among them is allowed. They also may not be recursive themselves.

2.2.1 Expressions and Types

Expressions are the core of OCL. Expressions can be used in various contexts, for example, to define constraints such as class invariants and pre, post conditions on operations [Ric02].

In the following section, we show all kinds of the OCL expressions based on the OCL 2.0 specification [OMG03].

Literal Expression

A Literal expression is an expression with no arguments producing a value. In general the result value is identical with the expression symbol. This includes *Integer*, *Real*, *Boolean*, *String*, *Collection*, *Tuple*, and *Enumeration* literals.

Integer Literal

An Integer Literal expression denotes a value of the predefined type *Integer*. For example, `-10, 5, 8`.

Real Literal

A Real Literal expression denotes a value of the predefined type *Real*. For example, `1.5, 6.78`.

Boolean Literal

A Boolean Literal expression represents the value `true` or `false` of the predefined type *Boolean*.

String Literal

A String Literal expression denotes a value of the predefined type *String*. For example, `'hello world', 'this is a string'`.

Collection Literal

A Collection Literal expression is a reference to a collection literal, denoting a collection of values of the same type. The elements of a collection are separated by commas, and surrounded by curly brackets. The kind of a collection preceeds the curly brackets. The kinds of collection literals are *Set*, *OrderedSet*, *Bag*, and *Sequence*.

A *Set* is the mathematical set. It does not contain duplicate elements. An *OrderedSet* is like a *Set* in which the elements are ordered. A *Bag* is like a set, which may contain duplicates. A *Sequence* is like a *Bag* in which the elements are ordered.

For example,

```
Set{3, 5, 7, 1}
OrderedSet{1, 3, 5, 7}
Bag{1, 2, 3, 1, 3}
Sequence{1, 3, 5, 5, 7, 9}
```

The type of a collection literal is simply *Set*(T), *OrderedSet*(T), *Bag*(T), or *Sequence*(T). T denotes the type of the elements of the collection. Those collection types have an abstract supertype *Collection*(T). The *Collection*(T) is used to describe the common properties of its subtypes.

The collection literal expressions in the above examples have types *Set*(*Integer*), *OrderedSet*(*Integer*), *Bag*(*Integer*), and *Sequence*(*Integer*) respectively.

OCL provides another alternative to write a sequence of integers through an interval specification. The interval specification consists of two expressions of type *Integer*,*int-expr*$_1$ and *int-expr*$_2$, separated by '..'. It denotes all the integers between the values of *int-expr*$_1$ and *int-expr*$_2$, including the values of *int-expr*$_1$ and *int-expr*$_2$ themselves. Therefore, the expressions

```
Sequence{ 1..(2*3) }
Sequence{ 1..6 }
```

are both identical to

```
Sequence{ 1, 2, 3, 4, 5, 6}
```

OCL 2.0 allows nested collections. This is different from OCL 1.4 where collections were always implicitly flattened. Instead, OCL 2.0 provides *flatten* operation to enable one to flatten a nested collection explicitly [OMG03].

There are many predefined operations for collection types. Operations which are common for the *Set*(T), *OrderedSet*(T), *Sequence*(T), and *Bag*(T) are defined under their supertype *Collection*(T). The common operations for all the collection types are *size*, *includes*, *excludes*, *includesAll*, *excludesAll*, *count*, *isEmpty*, *notEmpty*, and *sum* [OMG03].

Besides the common operations, there are also operations which are specific to certain collection types defined in the OCL 2.0 Standard Library [OMG03].

The predefined operations which are useful to transform a set into a bag or sequence, a bag into a set or sequence, and a sequence into a set or bag are *asSet*, *asBag*, and *asSequence* respectively.

For example, the expression `Set{ 3, 3, 5, 7, 1 }` is semantically not correct because a set may not contain duplicated elements. This expression can be corrected using the operation `asSet` as follows:

`Set{ 3, 3, 5, 7, 1 }.asSet()`, which evaluates to `Set{ 3, 5, 7, 1 }`.

OCL also defines a number of operations to handle the elements of a collection. These operations iterate over every element in a collection and evaluate an expression for each. The different constructs of the operations are shown in Section 2.2.1.

Tuple Literal

The concept of tuple is added to OCL 2.0 to enable full use of OCL as a query language. A tuple literal consists of named parts, each having a label, an optional type, and a value. The parts are separated by commas and is enclosed in curly brackets.

We construct an OCL tuple literal by enumerating the parts preceeded with the keyword `Tuple`.

For example,

```
Tuple {name: String = 'John',  age: Integer = 25}
```

The order of the parts is not important. Therefore, the following tuple literals are equivalent.

```
Tuple {name: String = 'John',  age: Integer = 25}
Tuple {name = 'John',  age = 25}
Tuple {age = 25, name = 'John'}
```

The OCL tuple literal is generally known as *record* or *struct*, while the named part is known as record *field*. A tuple literal has a tuple type of the form: *Tuple*(l_1:t_1,...,l_n:t_n),

where l_i denotes the label name, t_i denotes the type; $i \in 1..n$.

The main operation of a tuple type is the projection of a tuple value into one of its components. The dot notation followed by the label name projects a tuple value to the component of that label name.

For example,

```
Tuple {name: String = 'John',  age: Integer = 25}.age = 25
```

Enumeration Literal

An Enumeration Literal expression represents a reference to an enumeration literal. An enumeration defines a set of literals, and gives a name to it. For example, we can refer to an enumeration literal in Figure 2.1, by writing the following expression:

```
CourseType:: preliminary
```

This expression refers to the value `preliminary` of the enumeration type `CourseType`.

The set of the literals forms an enumeration type of the respective name. Enumeration types are user defined types.

For example, by using our class diagram example in Section 2.1, we can write the following invariant.

```
context Teacher
inv: maritalstatus = MaritalStatus::single
```

The invariant states that the marital status of every teacher is single.

On the enumeration types we can perform many of the common operations, such as testing for equality and inequality of two enumeration values, testing for the undefined value (see Section 2.2.2).

The OCL standard library defines several enumeration types which allow the modeler to refer to elements defined in the UML model. Those model element types are:

- *OclModelElement*, for each element in a UML model there is a corresponding enumeration literal.
- *OclType*, for each Classifier in a UML model there is a corresponding enumeration literal.
- *OclState*, for each State in a UML model there is a corresponding enumeration literal.

Let Expression

A Let expression consists of one or more variable declarations and a body expression (*in-expression*). The variable defined in the declaration is visible in the *in-expression*. A Let expression allows us to define a sub- expression which is used more than once in a constraint.

We use a Let expression to define a new variable with a type and a value. The variable defined by a Let expression cannot change its value. In the earlier versions of OCL, we can use a Let expression to define functions, however it is no longer allowed in OCL 2.0 [OMG03]. Instead, we can define a helper function using the *definition* constraint (see Section 2.2).

For example,

```
context Student inv:
  let schools : Integer = self.universities->size() in
  if isActive then
     schools >= 1
  else
     schools < 1
  endif
```

The Let expression declares the variable `schools` with has a value, the number of universities where a student is registered. This variable is used twice in the body expression to define an invariant.

Variable Expression

A Variable expression is an expression which consists of a reference to a variable. For example, references to the variables *self* and *result*, and the variables defined by the Let expressions.

If Expression

An If expression is composed of a *condition*, *then-expression*, and *else-expression*. This expression evaluates to one of two alternative expressions depending on the evaluated value of then *condition*. If the *condition* evaluates to *true*, the *then-expression* is evaluated, otherwise *else-expression* is evaluated. Both the *then-expression* and the *else-expression* are mandatory.

For example, referring to the example in Section 2.2.1: the invariant states that if the status of a student is active then the number of his/her schools should be greater than or equal to 1, and less than 1 otherwise.

Model Property Call Expression

A model property call expression is an expression that refers to a property (operation, attribute, association end) of a *Classifier* in a UML model to which the expression is attached. Its result value is the evaluation of the corresponding property.

There are three kinds of model property call expresssions, namely: *attribute call*, *operation call*, and *navigation call*.

Attribute Call Expression

An attribute call expression is a reference to an attribute of a *Classifier* defined in a UML model. It evaluates to the value of the attribute.

To refer to the attribute `age` of a certain class, one can write: `self.age`. For example, the expression `self.age` evaluates to the value of the `age` attribute of the particular instance of `Student` identified by `self`, as in:

```
context Student
inv: self.age  > 0
```

The invariant states that the age of a particular instance of the class `Student` should be greater than 0.

Operation Call Expression

An operation call expression refers to the query operation defined in a classifier. A query operation means an operation without side-effect. In a class model, an operation is defined to be side-effect-free if the *isQuery* attribute of the operations is true.

If the operations have parameters, the expressions will contain a list of arguments expressions. The number and types of the arguments must match the parameters. To refer to an operation that does not take parameters, parentheses with an empty argument list are mandatory. For example, to refer to the operation `tuitionFee` of the class `Student`, one may write:

```
context Student
inv: self.tuitionFee()  > 0
```

The expression `self.tuitionFee()` invokes the operation `tuitionFee` on a particular instance of the class `Student`. This operation call returns the tuition fee of a student, and of type *Real*, see Figure 2.1. The invariant states that the tuition fee of a student must be greater than 0.

Navigation Call Expression

In Section 2.2.1 we show the examples of collection literals. In this section we show another way to obtain a collection through a navigation call expression.

A navigation call expression is a reference to an association end or an association class defined in the UML model. It is used to determine objects linked to a target object by an association.

We can refer to the other objects and their properties in the class diagram by navigating to the opposite association end.

```
object.associationEndName
```

If the multiplicity of the association-end has a maximum of one ("0..1" or "1"), then the value of the expression is an object, otherwise the value is a set of objects on the other side of the `associationEndName` association.

For example,

```
context Department
inv: self.director.isUnemployed = false
inv: self.member->notEmpty()
```

In the first invariant, the navigation call expression `self.director` evaluates to a `Teacher` because the multiplicity of the association is "1". Consequently, we can navigate the attribute `isUnemployed` of the class `Teacher`. The invariant states that the unemployment status of the director of a department should be `false`.

In the second invariant, the navigation call expression `self.member` evaluates to a set of teachers, because the association end `member` has multiplicity greater than 1. Therefore, we can apply the collection operation `notEmpty` to this expression. The invariant is *true* if the department has at least one member.

An important point to note is that when we navigate through more than one association with multiplicity greater than 1, we get a bag instead of a set [WK99].

For example,

```
context Teacher
def: numberOfStudents = self.teaches.student->size()
```

The above example results in a bag of students because a student can participate in many courses taught by a teacher. If we want to count the number of students of a teacher using the above definition, we may end up with the wrong number because a student may occur twice in the collection.

Fortunately, we can use the OCL standard operation to transform a bag into a set. Using this operation, we can correct the definition:

```
context Teacher
def: numberOfStudents = self.teaches.student->asSet()->size()
```

When the association on the class diagram is adorned with { `ordered` }, the navigation results in an ordered set.

Loop Expresssion

A loop expression is an expression that represent a loop construct over a collection. It has an iterator variable to represent an element of the collection. The body expression is evaluated for each element in the collection. The result of a loop expression depends on the specific kind of collection and operation.

There are two kinds of the loop expressions, namely the *iterate* expression and the *iterator* expression.

The *iterate* expression is an expression which calls the *iterate* operation. This operation results in one value which is built from the evaluated value of the body expression over every elements of a collection [1]. The result can be of any type defined by the result variable.

The syntax of *iterate* is as follows:

```
collection->iterate(elem:Type; result:Type = <expression> |
                    body-expression )
```

The variable `elem` is the iterator, while the variable `result` is the accumulator. The accumulator gets an initial value `expression`. When the iterate is evaluated, `elem` iterates over the `collection` and the `body-expression` is evaluated for each `elem`. After each evaluation of `body-expression`, its value is assigned to `result`. The value of `result` is built up during the iteration of the collection.

For example, in the context of the university:

[1] The *iterate* operation is similar to the function *fold* in Haskell.

```
context University
inv: self.students->iterate( s:Student; result:Boolean=true |
                                    result and s.age >= 18 )
```

The scope of the variables `s` and `result` is the `body-expression`. The `body-expression` is evaluated on all instances of the students, and the result is saved in the variable `result`. The expression will return *true* if all instances of the students have an age greater than or equal to 18.

The *iterator* expression represents all other predefined collection operations that use an iterator. It includes *forAll*, *exists*, *select*, *reject*, and *collect*. The complete list of the iterator operations can be found in the OCL 2.0 specification [OMG03], pages 6-16 to p.6-19.

The examples of the iterator expressions are shown below:

Exists Operation

The *exists* operation is used to specify a boolean expression which must hold for at least one element in a collection. The *exists* operation can be written in one of the following forms:

```
collection->exists( boolean-expression )
collection->exists( v | boolean-expression-with-v )
collection->exists( v : Type | boolean-expression-with-v )
```

The *exists* operation will result in *true* if the boolean expression evaluates to *true* for at least one element of the collection.

The variable `v` ranges over the collection. The type annotation `Type` is optional since the type of the variable `v` is equal to the element type of the `collection`.

The scope of the boolean expression is the element of the collection on which the *exists* is invoked.

For example, in the context of the university:

```
context University
inv: self.students->exists( age < 18 )
inv: self.students->exists( s | s.age < 18 )
inv: self.students->exists( s:Student | s.age < 18 )
```

The expression `self.students` evaluates to a set of students. These three expressions have identical meaning and they evaluate to *true* if the `age` of at least one student in the set is less than 18.

ForAll Operation

The *forAll* operation is used to specify a boolean expression which must hold for all elements in a collection. Similar to *exists*, the *forAll* operation can be written in one of the following forms:

```
collection->forAll( boolean-expression )
collection->forAll( v | boolean-expression-with-v )
collection->forAll( v:Type | boolean-expression-with-v )
```

For example, in the context of the university:

```
context University
inv: self.students->forAll( age <= 40 )
inv: self.students->forAll( s | s.age <= 40 )
inv: self.students->forAll( s:Student | s.age <= 40 )
```

These expressions evaluate to *true* if the `age` of all students are less than or equal to 40.

Select Operation

The *select* operation enables one to select a subset of a collection by specifying a boolean expression which will be evaluated on each element of the colletion. The result of this operation is a collection that contains all the elements of the source collection for which the boolean expression evaluates to *true* [2].

The *select* operation can be written in one of the following forms:

```
collection->select( boolean-expression )
collection->select( v | boolean-expression-with-v )
collection->select( v:Type | boolean-expression-with-v )
```

For example, in the context of the university:

```
context University
inv: self.students->select(age >= 18)->notEmpty()
inv: self.students->select(s | s.age > 18)->notEmpty()
inv: self.students->select(s:Student | s.age > 18)->notEmpty()
```

[2]The *select* operation is similar to the function *filter* in Haskell.

The expression `self.students` evaluates to a set of students. The *select* operation takes each student from the set and evaluates `age >= 18` for this student. If it evaluates to *true*, then the student is included in the result set. The invariant will be *true*, if the resulting collection is not empty.

Reject Operation

The *reject* operation is identical to the *select* operation, but with *reject* operation we get a subset of a collecton for which the boolean expression evaluates to *false*.

For example, in the context of the university:

```
context University
inv: self.students->reject( age < 18 )->isEmpty()
```

The expression `self.students` evaluates to a set of students. The *reject* operation takes each student from the set and evaluates `age < 18` for this student. If it evaluates to *false*, then the student is included in the result set. The invariant will be *true*, if the resulting collection is not empty.

The *reject* operation can be expressed using *select* operation by specifying the negated boolean expression. For example, the above expression can also be written as:

```
context University
inv: self.students->select( not (age < 18) )->isEmpty()
```

Collect Operation

We have seen that the *select* and *reject* operations always result in a sub-collection of the original collection. If we want to compute a collection from another collection, then we can use *collect* operation [3].

The *collect* operation can be written in one of the following forms:

```
collection->collect( body-expression )
collection->collect( v | body-expression-with-v )
collection->collect( v:Type | body-expression-with-v )
```

The result of this operation is the collection of the evaluation results of the body expression on each element of the collection.

For example, to specify the collection of the last names of all students in the context of the university, one can write:

[3]The *collect* operation is similar to the function *map* in Haskell.

```
self.students->collect( lastName )
self.students->collect( s | s.lastName )
self.students->collect( s:Student | s.lastName )
```

The first form is the simplest one. Since the scope of the body expression is the element of the collection on which the *collect* is invoked, the property *lastName* is taken in the scope of a student.

In the second form the iterator variable `s` represents an element of the collection, which is a student, in this example. The body expression is qualified by the variable, and evaluates to the last name of a student. The third form is similar to the second one, except that the iterator variable is annotated with its type.

The expressions results in a collection of last names. The resulting type is not a set, but a bag because several students can have the same last names. The bag resulting from the collect operation always has the same size as the original collection.

The iterate expression is basic/elementary in that all other iterator operations can be written as iterate expressions. The following examples show how the various *iterator* operations can be expressed using the *iterate* operation:

collection → forAll(v | *boolean-expr-with-v*)
≡ collection → iterate(v ; $result$: Boolean = true |
$result$ and *boolean-expr-with-v*)
collection → exists(v | *boolean-expr-with-v*)
≡ collection → iterate(v ; $result$: Boolean = false |
$result$ or *boolean-expr-with-v*)
collection → select(v | *boolean-expr-with-v*)
≡ collection → iterate(v ; $result$= collection |
if *boolean-expr-with-v* then $result$
else $result$ → excluding(v) endif)
collection → reject(v | *boolean-expr-with-v*)
≡ collection → iterate(v ; $result$= collection |
if *boolean-expr-with-v* then $result$ → excluding(v)
else $result$ endif)
collection → collect(v | *expr-with-v*)
≡ collection → iterate(v ; $result$: T = $kind${ } |
$result$ → including(*expr-with-v*))

The type `T` in the *iterate* operation for *collect* is a collection type of certain *kind* (*Set, OrderedSet, Bag, Sequence*) which has element type equal to the type of *expr-with-v*. The initial value of *result* is an empty collection of the same *kind*. The *kind* of the collection is equal to the kind of the initial collection *collection*.

Variants of the OCL Expressions

OCL allows some variants in writing the language constructs, mostly shorthands intended to simplify the specification of the expressions. We should consider these shorthands when processing OCL expressions.

Implicit Model Property Call

If we want to refer to a property (attribute, operation, association) of a classifier, we usually write

```
classifierName.propertyName or
classifierName.propertyName(par1, par2, ...)
```

where the latter refers to a parameterized property.

However, in OCL we are allowed to refer to a property without specifying the classifier name. So that when we write

`propertyName` or `propertyName(par1, par2, ...)`,

it implicitly refers to the property of the current class or its superclass.

Shorthand for Collect

If we want to collect a property of a classifier, we normally use the iterator expression `collect`. For example,

```
self.students->collect(firstName)
```

By using shorthand notation for *collect*, we can write it as a navigation call through the object, as follows:

```
self.students.firstName
```

In general, if we apply a property to a collection of objects, it will automatically be interpreted as a *collect* over the members of the collection with the specified property [OMG03].

Therefore, for any property that belongs to the objects of a collection, the following two expressions are identical:

```
collection.propertyname
collection->collect(propertyname)
```

and similarly for the parameterized property:

```
collection.propertyname (par1, par2, ...)
collection->collect (propertyname(par1, par2, ...))
```

Iterator Variables

The collection types have a number of predefined iterator operations, such as `forAll, exists, select, reject, collect` and `iterate`. When we call the iterating methods, we can specify the iterator variable. The iterator variable refers to an object in the collection. In general, there are three possible forms for writing the iterator operations, namely:

```
collection->iterMethod( expression )
collection->iterMethod( v | expression-with-v )
collection->iterMethod( v:Type | expression-with-v )
```

The *forAll* operation has an extended variant in which multiple iterators are allowed. Those iterators will iterate over the complete collection and must be of the same type. For example,

```
context University inv:
self.students->forAll( s1, s2:Student |
               s1 <> s2 implies s1.lastName <> s2.lastName)
```

Accumulator Variable

The *iterate* operation requires an accumulator variable, which should have an initial value.

```
context University inv:
self.students->iterate( s:Student; acc:Boolean = true |
                            acc and s.isActive )
```

After each evaluation of the body expression `acc and s.isActive`, the value is assigned to the accumulator `acc`. Therefore, the value of `acc` is built up during the iteration over the collection.

2.2.2 The Type Hierarchy

OCL types are organized in a hierarchy. The type hierarchy shown in Figure 2.2 is based on the description in OCL 2.0 specification [OMG03].

The type hierarchy depicts the subtype relation (also known as *type conformance*) between types. The subtype relations defined in OCL are:

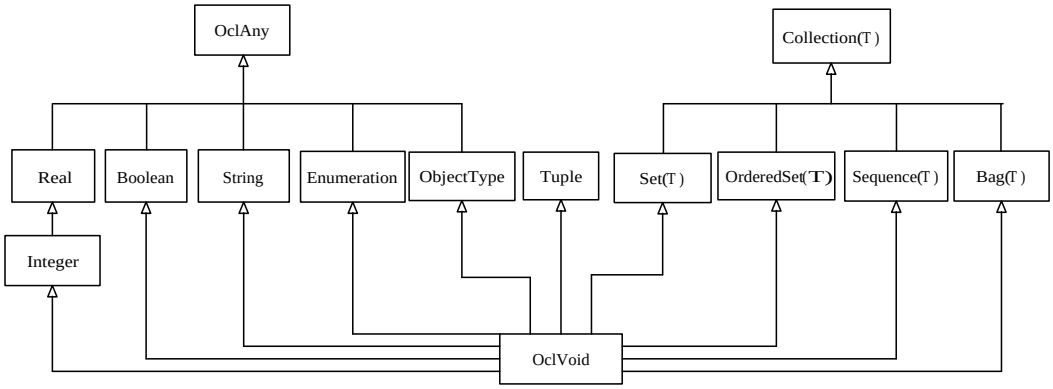


Figure 2.2: *OCL types hierarchy*

- *Integer* is a subtype of *Real*,
- All types, except the collection and tuple types, are subtypes of *OclAny*.
- *OclVoid* is a subtype of all other types,
- *Set*(*T*), *OrderedSet*(*T*), *Sequence*(*T*), and *Bag*(*T*) are subtypes of *Collection*(*T*).
- The hierarchy of types introduced by UML model elements mirrors the generalization hierarchy in the UML model.

OCL also includes polymorphism features. Polymorphism is the ability of a program fragment to have multiple types [Car97]. Polymorphism can be categorized into *parametric* and *ad-hoc* polymorphism [CW85]:

Parametric polymorphism

Parametric polymorphism is obtained when a function works uniformly on an infinite number of types which exhibit some common structure. A parametric polymorphic function will execute the same code for arguments of any admissible type. Parametric polymorphism is normally achieved by type parameters. Collection types in OCL are an example of *parametric polymorphism*. The collection type is parameterized with its element type.

Ad-hoc polymorphism

Ad-hoc polymorphism is obtained when a function works on different types. An ad-hoc polymorphic function may execute different code for each type of its argument.

There are two kinds of ad-hoc polymorphism, namely *overloading* and *coercion*.

In *overloading*, the same variable name is used to denote different functions. *coercion* on the other hand is a semantic operation which is needed to convert an argument to the type expected by a function.

The arithmetic operators ('+', '-', '/', '*') used in OCL are examples of *ad-hoc polymorphism*. By allowing thoses operators applied to both *Integer* and *Real*, the operator functions have overloaded meanings. When one of the arguments is of type Integer and the other is of type *Real*, then the *Integer* argument is coerced to the type *Real*.

Basic Types

Basic types in OCL include *Integer*, *Real*, *Boolean*, and *String*. OCL has a set of predefined operations on the basic types.

Table 2.1 shows examples of the predefined operations on the basic types.

Type	Operations
Integer	`*, +, -, /, abs()`
Real	`*, +, -, /, floor()`
Boolean	`and, or, xor, not, implies, if-then-else`
String	`concat(), size(), substring()`

Table 2.1: *Operations on basic types [OMG03]*

Object Types

Object types correspond to the classes of the UML models and are used to describe the set of possible object instances. An object type is defined such that it has the same name as the class name.

Operations on object types can be classified as follows:

- *Predefined operations*: operations which are implicitly defined in OCL for all object types.
- *Attribute operations*: accessing an attribute value of an object in a given system state.
- *Object operations*: accessing the operations of a class that do not have side effects. Those operations are marked with the tag *isQuery* in the UML model.
- *Navigation operations*: following an associaton link to retrieve the connected objects.

Special Types

OCL also provides some special types, namely *OclAny*, and *OclVoid*.

OclAny is the supertype of all other types except for the collection and tuple types. The common operations on all types are defined within *OclAny*. All its subtypes inherit all those operations.

The common operations on all types include equality test (=, <>), *oclIsUndefined*, *oclAsType*, *oclIsTypeOf*, *oclIsKindOf*, *allInstances*, and some other operations.

OclVoid is the subtype of all other types. The only predefined operation of *OclVoid* is *oclIsUndefined*.

2.2.3 Context Declarations

Besides the *classifier* context, OCL expressions may also be used with the other context declarations. In the following sections, we show the syntax of the declarations and examples of OCL expressions that are used with these contexts.

Operation Context

The operation context declaration is used for the expressions that can be coupled to the operation of a classifier. Those expressions include *precondition*, *postcondition*, and *body* expressions.

The syntax of the operation context declaration is as follows:

```
context Typename::operationName(param1:Type1,...):ReturnType
pre : -- some expression to constrain the parameters
post: -- some expression to constrain the result of the operation
body: -- some expression
```

Pre- and Postconditions

Pre- and postcondition are always specified within the context of an operation. A *precondition* must be true at the moment that the operation is going to be executed. The obligations are specified by postconditions. A *postcondition* must be true at the moment that the operation has just ended its execution [CW02].

For example, referring to the UML class diagram that we have defined for `University`, the operation `admitStudent` can be invoked on the `University` objects. We can specify the constraints that an implementation of the operation has to fulfill, such as

```
context University::admitStudent(student:Student)
pre  : student.isTypeOf(Student)
post : self.students = self.students@pre->including(student)
       and result = self.students
```

This constraint expressed two things: in the pre-clause it checks the provided argument is of the correct type, while in the post-clause, includes the `student` to the set of students and return a new set of students.

We use the keyword `@pre` to refer to the value of a property at the start of the operation by postfixing it after the property name. Therefore, the property `self.students@pre` refers to the value of the property `students` of the `University` at the start of the operation.

The reserved keyword `result` is a predefined variable that can be used for accessing the return value of an operation.

Body Expression

The *body* expression is an OCL expression acting as the body of an operation. Therefore the type of the *body* expression must conform to the result type of the operation.

Evaluating the body expression gives the result of the operation at a certain point in time.

```
context Student::activePrograms():Set(Program)
pre  : self.isActive = true
body : self.program->select(p|p.graduateDate.isEmpty())
```

The above example shows the use of the *body* expression together with the *precondition* in one operation context.

Attribute or Association Context

The attribute or association context declaration declares an attribute or an association end. Within the context declaration, we can specify the initial value or derivation rule of the respective attribute or association end.

The syntax is as follow:

```
context Typename::attributeName: Type
init: -- some expression representing the initial value

context Typename::assocRoleName: Type
derive: -- some expression representing the derivation rule
```

In the following example, we use OCL expressions to indicate the initial value and the derivation rule for the attribute `isActive` of the class `Student`.

```
context Student::isActive: Boolean
init: true
derive: if self.program->exists(p|p.graduateDate.isEmpty())
        then true else false endif
```

Chapter 3

Parsing OCL Expressions

The first step in processing OCL expressions is the transformation of the expressions into a syntax tree. This is done by a parser.

The parser takes OCL expressions as input and transforms those expressions into OCL abstract syntax tree (AST) according to the OCL 2.0 abstract grammar [OMG03]. During the parsing process, syntactical errors shall be reported. The type checking process takes this abstract syntax tree as input for further analysis.

We start this chapter by first giving an overview of parser combinators in Section 3.1. The implementation of OCL parsers by using Utrecht Parser Combinators (UPC) libraries is explained in Section 3.2. In the explanation, we assume the reader has knowledge about parsing and grammar constructions using EBNF notations.

3.1 Parser Combinators

When writing a parser, we may use a set of basic parsing functions. To create a more complicated parser however, we often have to combine a number of parsers. The functions that help us to combine parsers are called *parser combinators* [JS01]. Although the basic parsing functions do not combine parsers, they are usually also called parser combinators. Using parser combinators, we write parsers which closely resemble the grammar of language.

Our OCL parsers are implemented in functional language Haskell using the Utrecht Parser Combinators (UPC). The UPC are parser combinators libraries developed in Utrecht University by Doaitse Swierstra [Swi]. The parser combinators are also written in Haskell. Therefore, writing our parsers amounts to translating the grammar to a functional program.

The advantages of the parser combinators are:

- they allow us to write the parsers efficiently, and with little effort,
- they prevent us from running a separate program in order to generate a parser, and
- they have a mechanism to repair the errors.

3.1.1 Basic Parser Combinators

In this section, we introduce some parser combinators used in our parsers. The UPC combinators have names that depict the meaning so that they are easy to remember.

To illustrate the use of parser combinators, let us look at following EBNF grammar.

```
IfExp := 'if' Expression 'then' Expression
         'else' Expression 'endif'
```

The grammar states that an If expression, denoted by the non terminal `IfExp`, is composed of a keyword '`if`' followed by an expression, a keyword '`then`', an expression, a keyword '`else`', an expression, and a keyword '`endif`' at the end.

To construct a parser for the If expression, we need

- parsers for the keywords '`if`', '`then`', '`else`', and '`endif`',
- a parser for the `Expression`,
- a function that combines those parsers.

To parse a keyword, we use the parser combinator `pKey` defined in UPC. `pKey` takes a string as an argument and returns a parser that recognizes the string described by its argument. For example, the parser for the keyword '`if`' is `pKey "if"`.

Since `Expression` is a part of the grammar, we have to create a parser for it. The parser is not predefined in UPC libraries. Suppose we have created a parser for `Expression` called `pExpression`.

Having all the parsers created, the last step is to combine the parsers using parser combinators. The parser combinator `<*>` does the sequential composition where it combines two parsers into a single parser. The first parser returns a function, the second parser a value, and the combined parser returns the value that is obtained by applying the function to the value

The parser for the If expression will look like:

```
pIfExp = pSucceed f
     <*> pKey "if"   <*> pExpression
     <*> pKey "then" <*> pExpression
     <*> pKey "else" <*> pExpression
     <*> pKey "endif"
```

Since the combinator `<*>` expects a parser that returns a function as its left argument, we add the parser `pSucceed f`. This parser returns the function `f`.The combinator `pSucceed` is a special combinator which always succeeds. Given a value, it will return a parser of that value.

Besides for sequential composition, we also need a parser for choice. For this, we have the parser combinator `<|>`. Let us take the grammar for the boolean literal expression as an example.

```
BooleanLiteral := 'true' | 'false'
```

The grammar states that a boolean literal value is either '`true`' or '`false`'.

The parser for the boolean literal expression is straightforward given the combinator:

```
pBooleanLiteral = pKey "true" <|> pKey "false"
```

Although we are able to parse all the constructs in our grammar, we often need to postprocess the result value. In this case, we can use a new parser combinator: `<$>`. This combinator takes a function and a parser as its arguments, and applies the function to the result of the parser. Although the combinator `<$>` is derived from `<*>`, we refer to it as a basic parser combinator.

For example, we may need a parser for a boolean literal that recognizes a string '`true`' or '`false`', but returns the result as an integer (1 or 0 respectively), instead of a string. The modified version of the parser from the previous example is as follows:

```
pBooleanBit =  f <$> ( pKey "true" <|> pKey "false" )
  where f s = if (s == "true") then 1 else 0
```

The parser `pBooleanBit` recognizes the same string as `pBooleanLiteral`, but postprocesses the result using the function `f`. The function `f` is applied to the result value of the parser `pKey "true"` or `pKey "false"`, which is a string in this example.

We have now the knowledge of basic parser combinators. There are many variants of combinators which are derived from the basic ones. We will discuss those combinators in Section 3.2 by using the implementation of OCL parsers as the examples.

3.1.2 Error Recovery

We mentioned in the previous section that the UPC has a mechanism for error recoveries. In this section we list the kinds of the recoveries and show them in some examples. The recovery processes in the UPC have been taken in some precautions so that they will not derail [DS03].

If there are errors found, the parser will generate a list of error messages. The error messages indicate the corrections made to the input. There are two kinds of correcting steps:

- Insertion step, that insert a symbol into the stream of input symbols.
- Deletion step, that remove a symbol from the input stream.

The error recovery mechanism in the UPC enables the process to continue after the corrections.

For example, suppose we have a parser for the If expression according to the grammar in Section 3.1. Given a file containing the following If expression as input:

```
if true then 1 else 0
```

Since the keyword `endif` is missing in the above expression, the parser will repair it by inserting the keyword. The correction that has been made is reported as error messages as follows:

```
Error      : at end of file
Expecting  : symbol endif
Repaired by: inserting: symbol endif
```

On the other hand, if the parser encounters an unexpected symbol, it will repair the error by deleting that symbol. By using the same example as the previous one, but this time there is a typo at the keyword `endif`.

```
if true then 1 else 0 endiff
```

The parser does some correcting steps and produces the following error messages:

```
Error      : at lower case identifier endiff
Expecting  : symbol < or symbol <= or symbol ...
Repaired by: deleting: lower case identifier endiff

Error      : at end of file
Expecting  : symbol endif
Repaired by: inserting: symbol endif
```

3.2 The Parser Implementation

In this section, we will explain the implementation of OCL parsers for some OCL constructs. The complete implementation of OCL parsers can be seen in appendix B. The understanding of the basic parser combinators explained in Section 3.1.1 is essential.

3.2.1 The Expressions

In this section, we will explain the parsers implementation for some OCL constructs. The concrete syntax of a particular OCL construct preceeds each explanation.

If Expression

Let us back to the gramar for the If expression: an If expression is composed of a keyword '`if`' followed by an expression, a keyword '`then`', an expression, a keyword '`else`', an expression, and a keyword '`endif`' at the end.

```
IfExp := 'if' Expression 'then' Expression
         'else' Expression 'endif'
```

The corresponding parser for the above grammar is as follows:

```
pIfExp :: Parser Token Expression
pIfExp = IfExp <$> pKeyPos "if" <*> pExpression
               <*  pKey "then"  <*> pExpression
               <*  pKey "else"  <*> pExpression
               <*  pKey "endif"
```

The parser is slightly different from the one in Section 3.1.1. Two new combinators are introduced: `pKeyPos` and `<*`. The combinator `pKeyPos` is a variant of `pKey` in which it parses a keyword, but returns the position of the keyword intead of the string.

When we parse a sequence of expressions, sometimes we are not always interested in all values returned by the parsers. In this condition, we can use special versions of `<*>`: `<*` and `*>`. We use the combinator `<*` if we are not interested in the value returned by the right hand side of a production once we have recognized the keyword. Whereas the combinator `<*` is used on the other way around.

In the case of the If expression, we are no longer interested in the keywords `then`, `else`, and `endif` after we parse it, therefore the combinator `<*` is used.

Finally, we apply the constructor function `IfExp` to the resulting parser using the combinator `<$>`. The application results in an abstract syntax tree (parse tree) of the If expression:

`IfExp` *pos condition-expression then-expression else-expression.*

The result of the parser `pIfExp` is of type `Expression` as denoted by the type signature.

Let Expression

A Let expression is composed of a keyword '`let`' followed by a sequence of variable declarations, a keyword '`in`', and an expression.

```
LetExp := 'let' VariableDeclaration (',' VariableDeclaration)*
          'in' Expression
```

The parser for the corresponding concrete grammar above is implemented as follows:

```
pLetExp :: Parser Token Expression
pLetExp = LetExp <$> pKeyPos "let"
                 <*> pList1Sep (pKey ",") pVariableDeclaration
                 <*  pKey "in"
                 <*> pExpression
```

To parse a Let expression, we deal with a repetition of variable declarations, denoted by the EBNF notation `*`. The combinator that corresponds to the `*` notation is `pList`. Given a parser for a construct, `pList` constructs a parser for zero or more occurrences of that construct. So that the EBNF expression `X*` is implemented by `pList pX`, where `pX` is the parser for `X`.

Another EBNF notation for a repetition is `+` which accepts one or more occurences of a construct. The corresponding combinator for `+` is `pList1`. The implementation is similar to the one for `pList`, but we write `pList1 pX` instead for the EBNF expression `X+`.

The two combinators for repetition described above still do not meet our needs because the repetition in our grammar expects a separator between the elements. The combinators `pListSep` and `pList1Sep` enable us to create the parser. These combinators take a parser for the separator and a parser for the language construct as their arguments, and produce a new parser. The result of the new parser is a list of a certain constructs. The separators are of no importance.

In the parser for Let declarations, we use the combinator `pList1Sep` instead of `pLisSep` because a Let expression should contain at least one variable declaration. Therefore, corresponding parser for the grammar `VariableDeclaration (',' VariableDeclaration)*` is

```
pList1Sep (pKey ",") pVariableDeclaration
```

where `pVariableDeclaration` is the parser for `VariableDeclaration`.

Arithmetic Expressions

In the previous example, we showed the use of the combinator with separator. Now, we introduce the combinators in which the separator has a meaning. These combinators are useful when we want to parse arithmetic or logical expressions where the separators are arithmetic or logical operators. The combinators that can be used are `pChainl` and `pChainr`, to parse left associative and right associative sequences/list of constructions respectively.

For example, the grammar for an additive expression states that it consists of a sequence of multiplicative expressions separated by additive operators.

```
AdditiveExp := MultiplicativeExp (AdditiveOp MultiplicativeExp)*
AdditiveOp  := '+' | '-'
```

Since the additive operators are left associative, we use the parser combinator `pChainl`. The corresponding parser created for the above grammar is as follows:

```
pAdditiveExp :: Parser Token Expression
pAdditiveExp =
   pChainl ((\(p,op) e -> BinaryExp p e op) <$> pAdditiveOp)
           pMultiplicativeExp

pAdditiveOp :: Parser Token (Pos, String)
pAdditiveOp = (\p -> (p, "+")) <$> pKeyPos "+"
           <|>(\p -> (p, "-")) <$> pKeyPos "-"
```

`BinaryExp` is the constructor function for a binary expression. It takes four arguments: a position, a left expression, an operator, and a right expression.

The combinators `pChainl` and `pChainr` take two arguments, namely a parser for the separators and a parser for the language construct. They expect that the parser for the separators returns a function, which is then used by chain to combine parse trees for the elements. In the `pChainl`, the operator is applied left-to-right, while in the case of `pChainr` is applied right-to-left.

An important thing to note is that the separators should have the same associativity. In the example for the additive expression, the additive operators `"+"` and `"-"` are both left associative.

3.2.2 The Context Declarations

Our parsers for OCL context declarations are able to parse the three kinds of context declarations described in Section 2.2 and Section 2.2.3.

The corresponding concrete syntax can be seen in Appendix A. The syntax has incorported the requirements that a *classifier* context can only contain an invariant or a definition, and that an *operation* context contain an pre, post or body declaration. For example, consider the concrete syntax for the *classifier* context.

```
ClassifierContext  := 'context' PathName InvOrDef+
InvOrDef           := 'inv' SimpleName? ':' Expression
                    | 'def' SimpleName? ':' DefExpression
DefExpression      := VarDecl '=' Expression
                    | Operation '=' Expression
```

The syntax states that a *classifier* context requires an `inv` or `def` stereotype. In the Dresden OCL Toolkit [Fin00], this requirement is implemented as part of the semantic analysis. In our implementation, it is implemented as part of the syntactic analysis by the parsers. Given the above syntax, the parsers are straightforward as follows.

```
pContextDeclaration :: Parser Token ContextDeclaration
pContextDeclaration =
        attrorassoc  <$>  ...
    <|> classcontext <$> pKeyPos "context"
                     <*> (pNameU <**> pPath )
                     <*> pList1 pInvOrDef
    <|> opercontext  <$> ...
   where
     attrorassoc pos nms tp exprs
       = ...
     classcontext pos nms decls
       = ClassifierContext pos nms decls
     opercontext pos nms params tp decls
       = ...

pInvOrDef :: Parser Token InvOrDef
pInvOrDef =
        Invariant <$> pKeyPos "inv"
                  <*> pMaybeName
                  <*  pKey ":"
                  <*> pExpression
```

```
    <|> VariableDef <$> pKeyPos "def"
                    <*> pMaybeName
                    <*  pKey ":"
                    <*> pVariableDeclarationDef
                    <*> pExpression
    <|> operdef <$> pKeyPos "def"
                <*> pMaybeName
                <*  pKey ":"
                <*> (pNameLU <**> pPath)
                <*> pParens (pListSep (pKey ",") pFormalParameter)
                <*> pMaybeReturnType
                <*  pKey "="
                <*> pExpression
  where
    operdef pos mb nms params tp e
      = OperationDef pos mb (init nms) (last nms) params tp e
```

The parser `pContextDeclaration` implements the three kinds of OCL context declarations: *attribute or association*, *classifier*, and *operation* contexts.

Chapter 4

OCL Type System

The OCL parser does the syntactic analysis on the OCL expressions. It checks the expressions against the OCL abstract grammar. The next step of our compilation process is to check the semantic consistencies. Our semantic analysis focuses on the type checking process since type errors are the most common semantic errors that can be found in the correctly parsed OCL expressions. Furthermore, the type information resulting from the type checking process are needed for the later compilation processes.

The type checker computes types of the identifiers and expressions within an OCL expression and checks them against the expected types. It will report the errors if there were type inconsistencies found, and return the computed types if there were no errors found.

In this chapter, we focus on the formalization of OCL Type System. Prior to the discussion of the OCL type sytem, we first describe the static check process in Section 4.1. In Section 4.2 we define the type environment needed for the formalization of the OCL type system. In Section 4.3 we define the syntax of OCL types. An overview of a subtype relation in given in Section 4.4. In Section 4.5 we define a set of type rules which forms the OCL type system. The use of the type rules can be seen in Section 4.6 where we construct the derivation trees for some OCL expressions. Finally, in Section 4.7 we explain the implementation of the type checker.

4.1 Static Checks

On the correctly parsed tree, we need to perform some static checks. The static checks are intended to detect static errors in the early stage before we start the type checking process.

We perform the following static checks on the OCL abstract syntax tree:

- Check the context declarations.
- Check that all variables are in the scope.
- Check that all model properties are in the scope.
- Check for duplicated variables.
- Check the use of keyword @pre
- Check the use of variables `self` and `result`.

Check the context declarations

A context declaration introduces a scope for OCL expressions. Within the scope of a certain context, we can write the OCL expressions. The context declarations refer to a classifier, an operation, an attribute, or an association end of a UML model. If the declared context does not exist in the imported UML model information, we report a static error.

Check that all variables are in the scope

There are several places in the abstract syntax tree where variables can enter the scope:

- Let expression.

  The variables declared in a Let expression are added to the scope. They are passed to the body expression where we check that the body expression uses only the variables which are in the scope.

- Loop expression.

  The iterator and result variables in a loop expression are added to the scope and then passed to their body expression. In the body expression, we check that it uses only the variables which are in the scope.

Check that all model properties are in the scope

The model properties include attributes, operations, and association ends. The names of the properties are added to the scope from the imported UML model information. The names are passed down into the tree where we check that the expressions use only the properties that are in scope.

Check the duplicated variables

Duplicated variables may be found in a Let expression where variables of the same name are declared. In the *forAll* iterator expression where more than one iterator variable is allowed, we also check for duplicated variable declarations.

Check the use of keyword @pre

We use the keyword @pre to refer to the value of a property at the start of the operation, as discussed in Section 2.2.3. This implies that the keyword @pre can only be used in the operation context. If the keyword @pre is used in the classifier context or attribute/association context, a static error will be generated.

Check the use of variables `self` *and* `result`.

The variable `self` refers to the instance of the current context. While the variable `result` is used to access the return value of an operation. Therefore, the variables `self` and `result` may not be redefined, either as a Let variable or an iterator.

4.2 Type Environment

The type environment carries typing assumptions about variables. We use symbol Γ to refer to the environment.

The assumptions on variables can be grouped as follows:

- Assumptions on *term variables*, namely variables which are introduced by OCL expressions. An assumption about a term variable is represented by a pair (`name`, *Type*). A term variable is written as a lower case identifier. For instance, suppose we have the following Let expression:

  ```
  context Person
  inv: let income : Integer = self.job.salary->sum()
       in income > 1000
  ```

  The Let declaration will add a *term variable* binding to the environment, namely ("`income`", *Integer*).

- Assumptions on *type variables*, namely variables which are introduced by the polymorphic types, such as collection types. Our assumptions on the type variables also carry the subtype constraints; it is of the form $T_1 \leq T_2$. The type variables are represented as upper case identifiers.

We distinguish three kinds of the type environment. The syntax is shown in Figure 4.1.

$$
\begin{array}{lll}
\Gamma & := (\ \Gamma_{\mathcal{G}}, \Gamma_{\mathcal{C}}, \Gamma_{\mathcal{L}}\ ) & \textit{type environment} \\
\\
\Gamma_{\mathcal{G}} & := & \textit{global environment} \\
 & (\ T_1 \leq T_2\ )^* & \quad \textit{type variable binding} \\
\\
\Gamma_{\mathcal{C}} & := & \textit{class environment} \\
 & (\ T_1 \leq T_2\ )^* & \quad \textit{type variable binding} \\
\\
\Gamma_{\mathcal{L}} & := & \textit{local environment} \\
 & (\ x : T\ )^* & \quad \textit{term variable binding}
\end{array}
$$

Figure 4.1: *Abstract syntax of the type environment*

4.2.1 Global Environment

The global environment $\Gamma_{\mathcal{G}}$ carries assumptions about *types variables*. Our initial global environment contains the following information:

UML Model Information

The UML model information stored in the global environment are:

- Types of the classes, including their attributes, operations, and associations.
- The subtype relations between classes in a model.

Using the class diagram example in Section 2.1, the class `Student` and its properties will be represented in the global environment as follows:

$$
\begin{array}{ll}
Student \leq \{ & firstName : String, \\
 & lastName : String, \\
 & age : Integer, \\
 & gender : Gender, \\
 & isActive : Boolean, \\
 & tuitionFee : () \ \rightarrow \ Real, \\
 & birthdayHappens : () \ \rightarrow \ OclVoid, \\
 & universities : Set(University)\}
\end{array}
$$

The first five fields are the attributes of the class *Student*, while *tuitionFee* and *birthdayHappens* are the operations of the class, and *universities* is an

association end to the class *University*. We use collection type *Set* to represent the association end with multiplicity *. The arguments of an operation are represented by a product type. If an operation takes no arguments, we represent it as an empty product type, written as ().

The only subtype relation between classes in the diagram is

$$MasterStudent \leq Student.$$

Predefined Operations on All Objects

OCL primitive types and types in a UML model are subtypes of the *OclAny* that they inherit all the operations defined on *OclAny*. The instances of *OclAny* are called *object*. The predefined operations on all objects in the global environment are defined as follows:

$$\begin{aligned} OclAny \leq \{ = &: OclAny \to Boolean, \\ <> &: OclAny \to Boolean, \\ &oclAsType : OclType \to T, \\ &oclIsTypeOf : OclType \to Boolean, \\ &oclIsKindOf : OclType \to Boolean\} \\ &allInstances : () \to Set(T)\} \end{aligned}$$

Since the equality and inequality operators "=" and "<>" are common on all objects, they are defined as the properties of *OclAny*. This implies that the expressions "$e_1 = e_2$" or "$e_1 <> e_2$" are always legal. However, there is still a restriction for these expressions which states that e_1 and e_2 must be of the same type.

OCL Predefined types and their operations

OCL predefined types are defined as objects with properties. Whereas the properties are standard operations defined on those types. The assumptions on the predefined types and their operations are defined in the global environment as type conformances between the types and their properties. The properties of each type are collected in a record with the operation names and their type signatures as the record fields.

The access to a certain operation is through record projection. For example, the expression $s_1.union(s_2)$ invokes the operation *union* on the source expression s_1 with s_2 as the parameter to the operation. The type of s_1 should conform to the record containing operation *union*.

In the following subtype relations, we show how OCL predefined types and their operations are defined in the global environment.

$$
\begin{aligned}
Real \leq \{ \quad & <: Real \to Boolean, \\
& >: Real \to Boolean, \\
& + : Real \to Real, \\
& - : Real \to Real, \\
& * : Real \to Real, \\
& / : Real \to Real\}
\end{aligned}
$$

$$
\begin{aligned}
Integer \leq \{ \; & abs : () \to Integer, \\
& div : Integer \to Integer, \\
& mod : Integer \to Integer\}
\end{aligned}
$$

$$
\begin{aligned}
Boolean \leq \{ & or : Boolean \to Boolean, \\
& xor : Boolean \to Boolean, \\
& and : Boolean \to Boolean, \\
& not : Boolean, \\
& implies : Boolean \to Boolean\}
\end{aligned}
$$

$$
\begin{aligned}
String \leq \{ \; & size : () \to Integer, \\
& concat : String \to String, \\
& substring : Integer \to Integer \to String, \\
& toInteger : () \to Integer, \\
& toReal : () \to Real\}
\end{aligned}
$$

$$
\begin{aligned}
Collection(T) \leq \{ & size : () \to Integer, \\
& includes : T \to Boolean, \\
& includesAll : Collection(T) \to Boolean, \\
& isEmpty : () \to Boolean, \\
& sum : () \to T\}
\end{aligned}
$$

$$
\begin{aligned}
Set(T) \leq \{ \; & union : Set(T) \to Set(T), \\
& intersection : Set(T) \to Set(T), \\
& difference : Set(T) \to Set(T), \\
& including : T \to Set(T), \\
& excluding : T \to Set(T), \\
& symmetricDifference : Set(T) \to Set(T), \\
& asSequence : () \to Sequence(T), \\
& asBag : () \to Bag(T) \; \}
\end{aligned}
$$

4.2.2 Class Environment

The class environment $\Gamma_{\mathcal{C}}$ carries assumptions about the current context, which is the context to which a certain OCL expression atttached. The class

environment is a subset of the global environment in the sense that all information in the class environment are also part of the global environment. For example, in the context of the class $University$, we will have the following information stored in the class environment.

$$\begin{aligned} University \leq \{\ & name : String, \\ & address : String, \\ & foundedYear : Integer, \\ & students : Set(Student), \\ & departments : Set(Department), \\ & admitStudent : Student \ \rightarrow \ OclVoid\ \} \end{aligned}$$

4.2.3 Local Environment

The local environment $\Gamma_{\mathcal{L}}$ carries assumptions about *term variables*. Local variables are only accessible within the context where they have been defined, such as:

- Variables that are introduced by a Let expression.
- The iterator variables in the loop expresssion.
- The variables *self* and *result*.

  The variable *self* refers to the current classifier, which is the context to which a certain OCL expression is attached.

  The variable *result* refers to the object returned by an operation. When the operation has no out or in/out parameters, the type of the variable *result* is the return type of the operation. For example, the type of the variable `result` in the following example is *Integer*.

  ```
  context Person::income(d:Date):Integer
  post: result = age * 1000
  ```

  When the operation does have out or in/out parameters, the return type is a tuple containing the parameters and the variable `result`. For example, the postcondition for the `income` operation with out parameter `bonus` may have the following form:

  ```
  context Person::income(d:Date, bonus:Integer):Integer
  post: result = Tuple {  bonus = ...
                        , result = ...
                       }
  ```

```
Type    := String                                        constant type
         | Kind '(' Type ')'                             collection type
         | '{' Fields? '}'                               record type
         | Type '→' Type                                 function type
         | '(' Types? ')'                                product type

Types   := Type (',' Type)*                              sequence of types

Field   := String ':' Type                               field of a record

Fields  := Field (',' Field)*                            sequence of fields

Kind    := 'Set' | 'OrderedSet' | 'Sequence'             collection kinds
         | 'Bag' | 'Collection'
```

Figure 4.2: *The syntax of OCL types*

In this example, the variable `result` has a record type with fields `bonus` and `result`.

Although `self` and `result` are not keywords in OCL, they may not be used as the name of a Let variable or an iterator.

4.3 The Syntax of OCL Types

Before type checking OCL expressions, we first define the syntax of OCL types as shown in Figure 4.2.

The non terminal `Type` consists of five alternatives which represent the constant, collection, record, function, and product types respectively.

The constant types include the primitive types, object types, enumeration types, and special types.

The kind of a collection type is identified by the `Kind`. A collection type consists of a kind and the type of its elements. For instance, *Set(String)*, represents a set with *String* as the type of its elements.

A record type consists of a set of field types separated by commas which might be empty. The OCL tuple type is represented as a record type. For instance, the tuple type *Tuple* { *name:String, age:Integer* } is represented as { *name:String, age:Integer* }.

A product type is used in the context of the function type to represent the argument types. For example, (*Int, String*) → *Boolean* represents a function type which has two arguments: the first has type *Integer*, the second has type *String*, and the return type is *Boolean*. If a function takes no argument, we represent its argument type as an empty product type, written as ().

4.4 Subtyping

Subtyping is one of the features that can be found in almost all object oriented programming languages. Subtyping is also called *subtype polymorphism*. A subtype relation is written as a statement of the form $A \leq B$, pronounced "A is a subtype of B" (or "B is a supertype of A"), which means that any term of type A can be used safely in the context where a term of type B is expected [Pie02].

The subtype relation in OCL is known as *type conformance*. The type conformance in OCL satisfies the following properties:

- *Reflexive*, a type always conforms to itself.
- *Transitive*, if type A conforms to type B, and type B conforms to type C, then type A also conforms to type C.
- *Antisymmetric*, if type A conforms to type B, and type B conforms to type A, then A and B are equivalent.

  The subtype relation in OCL is *antisymmetric*, which is not always the case in other languages. In OCL, two records are the same when one is a permutation of the other. The main operation on a record is projection, i.e. accessing a field of a record by using its name. Record projection is insensitive to the order of the fields. For example, the record projection

  { `name` = "`John`", `age` = `25` }.`age` will produce the same result as

  { `age` = `25`, `name` = "`John`" }.`age` namely 25.

4.5 Type System

In the following sections, we will give the formalization of OCL type system in the form of a set of type rules.

4.5.1 Auxiliary Definitions

Before we continue, we first give some auxiliary definitions.

Removing elements from the environment.

Before adding an element to the environment, sometimes we need to remove the previous references on the same variables. To remove a set of elements $\mathcal{V}$ from the environment Γ, we write $\Gamma \setminus \mathcal{V}$, where

$$\Gamma \setminus \mathcal{V} \;=\; \{\; (\mathsf{a} : \tau) \mid (\mathsf{a} : \tau) \in \Gamma \;,\; \mathsf{a} \notin \; \mathcal{V} \;\}$$

Domain of the environment.

When defining a type rule, we often encounter a pair of the form $\mathsf{x} : \tau$, where usually x is a variable and τ is a type. If we want to get a list of the first elements from the set of pairs Γ, we can write $dom(\Gamma)$ where

$$dom(\Gamma) \;=\; \{\; \mathsf{x} \mid (\mathsf{x} : \tau) \in \Gamma \;\}$$

Collection kinds.

In formalizing type rules for the collection types, we often have to present the kinds of the collection types. We introduce a symbol $\mathcal{K}$ to represent the set of the collection kinds, while *kind* represents an element of $\mathcal{K}$.

$$\mathcal{K} = \{\; \text{'}\mathit{Set}\text{'}, \text{'}\mathit{OrderedSet}\text{'}, \text{'}\mathit{Bag}\text{'}, \text{'}\mathit{Sequence}\text{'}, \text{'}\mathit{Collection}\text{'} \;\} \;,\; \mathit{kind} \in \mathcal{K}$$

Variable Declarations

A variable declaration declares a variable of a certain type and may have an initial value. It has the general form $v : t = \mathsf{e}$, read "The variable v has a type t and an initial value e".

The use of variable declarations in expressions are shown in Table 4.1, each having different constraints on the type annotations.

Expression	Type Annotation
Let expression	mandatory
Iterator expression	optional
Tuple Literal expression	optional

Table 4.1: *The Use of variable declarations in expressions*

The optional type annotation influences the formalized type rules for the expressions. Basically, we need two type rules, one for each possibility. In order to simplify the formalized type rules, we only formalize type rules for the case where the annotation is omitted. If the type annotation is provided in the expression where it is optional, an additional condition, namely that the assigned type must conform to the annotated type, should be satisfied.

4.5.2 Type Rules

Having defined the type environment and the type language, we now define the type rules. A type rule introduces a formal representation on how an expression should be typed. A collection of type rules is called a *type system*.

A type rule asserts the validity of certain judgments on the basis of other judgments that are already known to be valid [Car97].

All type rules have the following form:

$$\text{NAME} : \frac{\Gamma_1 \vdash \mathcal{A}_1 \ldots \Gamma_n \vdash \mathcal{A}_n}{\Gamma \vdash \mathcal{A}}$$

Each rule has a name so that we can refer to it later. A type rule consists of a number of *premise* judgments $\Gamma_i \vdash \mathcal{A}_i$ above a horizontal line and a single *conclusion* judgment $\Gamma \vdash \mathcal{A}$ below the line. When all of the premises are satisfied, the conclusion must hold. The number of premises may be zero.

The formalized type rules corresponding to the subtype relation in OCL are shown in Figure 4.3. The subtype rules consist of a collection of subtype judgments. A subtype judgment is of the form:

$$\Gamma \vdash \tau_1 \leq \tau_2$$

which states that a type τ_1 is the subtype of a type τ_2 if the information is in the type environment Γ.

The main rule corresponds to the subtype relation is the rule S-SUB, which states that we can conclude that one type is a subtype of the other if that fact can be deduced from the environment. The environment is either a class or global environment. We first look up the subtype relation in the class environment. Only if the subtype relation is not in the class environment, we look it up in the global environment. The local environment contains no subtype relation information as depicted in the syntax of the type environment in Figure 4.1.

The rule S-REF expresses the reflexivity in the subtype relation, while the rule S-TRANS expresses the transitivity.

The subtype relation between *Integer* and *Real* means that an *Integer* value can be used where a *Real* value is expected.

The special types in OCL, namely *OclAny* and *OclVoid* influence the formalization of type rules by introducing a maximum and minimum type respectively.

OclAny is the supertype of all of the OCL types, except for the collection and record types. The exception is to simplify the type system and also to

$$\text{S-SUB}: \frac{\tau_1 \leq \tau_2 \in \Gamma}{\Gamma \vdash \tau_1 \leq \tau_2}$$

$$\text{S-REF}: \ \Gamma \vdash A \leq A$$

$$\text{S-TRANS}: \frac{\Gamma \vdash A \leq B \quad \Gamma \vdash B \leq C}{\Gamma \vdash A \leq C}$$

$$\text{S-INT}: \ \Gamma \vdash \mathit{Integer} \leq \mathit{Real}$$

$$\text{S-ANY}: \ \Gamma \vdash T \leq \mathit{OclAny}$$

$$\text{S-VOID}: \ \Gamma \vdash \mathit{OclVoid} \leq T$$

$$\text{S-SET}: \ \Gamma \vdash \mathit{Set}(T) \leq \mathit{Collection}(T)$$

$$\text{S-ORDSET}: \ \Gamma \vdash \mathit{OrderedSet}(T) \leq \mathit{Collection}(T)$$

$$\text{S-BAG}: \ \Gamma \vdash \mathit{Bag}(T) \leq \mathit{Collection}(T)$$

$$\text{S-SEQ}: \ \Gamma \vdash \mathit{Sequence}(T) \leq \mathit{Collection}(T)$$

$$\text{S-ESET}: \frac{\Gamma \vdash T_1 \leq T_2}{\Gamma \vdash \mathit{Set}(T_1) \leq \mathit{Set}(T_2)}$$

$$\text{S-EORDSET}: \frac{\Gamma \vdash T_1 \leq T_2}{\Gamma \vdash \mathit{OrderedSet}(T_1) \leq \mathit{OrderedSet}(T_2)}$$

$$\text{S-EBAG}: \frac{\Gamma \vdash T_1 \leq T_2}{\Gamma \vdash \mathit{Bag}(T_1) \leq \mathit{Bag}(T_2)}$$

$$\text{S-ESEQ}: \frac{\Gamma \vdash T_1 \leq T_2}{\Gamma \vdash \mathit{Sequence}(T_1) \leq \mathit{Sequence}(T_2)}$$

$$\text{S-RCDWIDTH}: \ \Gamma \vdash \{\ l_i : {T_i}^{\,i \in 1..n+k}\ \} \leq \{\ l_i : {T_i}^{\,i \in 1..n}\ \}$$

$$\text{S-RCDDEPTH}: \frac{\text{for each i } \Gamma \vdash S_i \leq T_i}{\Gamma \vdash \{\ l_i : {S_i}^{\,i \in 1..n}\ \} \leq \{\ l_i : {T_i}^{\,i \in 1..n}\ \}}$$

$$\text{S-RCDPERM}: \frac{\{\ k_j : {S_j}^{\,j \in 1..n}\ \} \text{ is a permutation of } \{\ l_i : {T_i}^{\,i \in 1..n}\ \}}{\Gamma \vdash \{\ k_j : {S_j}^{\,j \in 1..n}\ \} \leq \{\ l_i : {T_i}^{\,i \in 1..n}\ \}}$$

Figure 4.3: *Type Rules for the subtype relation in OCL*

avoid the cyclic domain definitions in the subtype relation. For example, if *OclAny* were the supertype of *Set(OclAny)* [OMG03].

OclVoid is the subtype of all other types.

The type *Collection* is an abstract type, with the concrete types are its subtypes: *Set*, *OrderedSet*, *Bag*, and *Sequence*. The subtype relation between collection types of a certain kind are determined by the subtype relation of their element types.

There are three rules corresponding to the subtype relation between record types described in Benjamin C. Pierce's book [Pie02]. These rules also apply to the record type in OCL.

The subtype relation between record types states that a record with $n+k$ fields is a subtype of the record with n fields as long as the first n fields are the same for both records. For example, suppose we have a function that expects a parameter r_1 of type { `name`*:String,* `age`*:Integer* }. We can call the function by providing an argument r_2 of type { `name`*:String,* `age`*:Integer,* `address`*:String* } because the first two fields of r_2 are the same with the fields of r_1. Therefore, r_2 is a subtype of r_1, which means that r_2 can be used in the expressions where r_1 is expected.

The types of the fields in a record also influence its subtype relations: "two records are in the subtype relation as long as the types of each corresponding field in those two records are in the subtype relations". For example, the record type { `name`*:String,* `salary`*:Integer* } is a subtype of the record type { `name`*:String,* `salary`*:Real* } because types of each corresponding field are in the subtype relations: *String* $\leq$ *String*, *Integer* $\leq$ *Real*.

Finally, the record with permutated fields is a subtype of the original record, and vice versa.

Besides the subtype judgment, we need another kind of judgment for typing an OCL expression. This judgment relates a type to an expression and is of the form:

$$\Gamma \vdash \mathtt{e} : \tau$$

It asserts that the expression `e` has a type τ with respect to the type environment Γ.

Our type rules are syntax directed which means that for each expression, there is exactly one corresponding type rule (except for the variable and iterator rules, although it never happens that two rules apply).

We now consider the OCL type rules one by one. Type rules for the literal expressions are shown in Figure 4.4.

$$\text{TRUE}: \vdash \mathsf{true} : \mathit{Boolean}$$

$$\text{FALSE}: \vdash \mathsf{false} : \mathit{Boolean}$$

$$\text{INT}: \vdash \mathsf{i} : \mathit{Integer}, \text{if } \mathsf{i} \in \mathbb{Z}$$

$$\text{REAL}: \vdash \mathsf{i} : \mathit{Real}, \text{if } \mathsf{i} \in \mathbb{R}$$

$$\text{STRING}: \vdash \mathsf{i} : \mathit{String}, \text{if } \mathsf{i} \in \text{String}$$

$$\text{COLLECTION}: \frac{\Gamma \vdash \mathsf{item}_i : \tau_i \ ^{i \in 1..n} \quad \Gamma \vdash \tau_i \ ^{i \in 1..n} \leq \tau}{\Gamma \vdash \mathit{kind}\ \{\ \mathit{item}_1, \ldots, \mathit{item}_n\ \} : \mathit{kind}(\tau)}$$

$$\text{TUPLE}: \frac{\Gamma \vdash \mathsf{e}_i : \tau_i \ ^{i \in 1..n}}{\Gamma \vdash \mathsf{Tuple}\{\ \mathsf{v}_1 = \mathsf{e}_1, \ldots, \mathsf{v}_n = \mathsf{e}_n\ \} : \{\ \mathsf{v}_1 : \tau_1, \ldots, \mathsf{v}_n : \tau_n\ \}}$$

Figure 4.4: *Type rules for the literal expressions in OCL*

The first five rules correspond to the basic values in OCL. They are relatively simple, and basically state that every basic value has its corresponding type. For example, values true and false have type *Boolean*, while value 3 and 1.5 have type *Integer* and *Real* respectively. No additional information from the environment is needed for typing the basic values.

The rule for a collection literal expression is a bit more difficult since we have to find the common super type of all elements of the collection, before we can conclude the type of the expression. In the OCL 2.0 specification [OMG03], it is implicitly stated that empty collections have *OclVoid* as their elements types.

A tuple literal expression has a record type, which is composed from the types of all its elements labelled with their field names.

The type rules for the other OCL expressions are shown in Figure 4.5:

The type rule for variables consists of three rules which denote the look up order. We first look up a variable in the local environment $\Gamma_{\mathcal{L}}$ (rule L-VAR). If it is not in the local environment, then we look it up in the class environment $\Gamma_{\mathcal{C}}$ (rule C-VAR), and finally if the variable is not in the local environment and neither in the class environment, we look it up in the global environment $\Gamma_{\mathcal{G}}$ (rule G-VAR). If the variable v has type t in one of the environments then we can conclude that the variable v is of type t.

We can view the binary expression as projecting the binary operator out of the left expression and applying the operator to the right expression. For example, the expression `a + b` can be viewed as `a.+(b)`.

The rule for the binary operators states that if we can assign a type τ_1 to the left expression, and a type τ_2 to the right expression, the type τ_1 conforms

$$\text{L-VAR}: \frac{(\mathsf{v} : t) \in \Gamma_{\mathcal{L}}}{\Gamma \vdash \mathsf{v} : t}$$

$$\text{C-VAR}: \frac{(\mathsf{v} : t) \in \Gamma_{\mathcal{C}} \quad \mathsf{v} \notin dom(\Gamma_{\mathcal{L}})}{\Gamma \vdash \mathsf{v} : t}$$

$$\text{G-VAR}: \frac{(\mathsf{v} : t) \in \Gamma_{\mathcal{G}} \quad \mathsf{v} \notin dom(\Gamma_{\mathcal{L}}) \quad \mathsf{v} \notin dom(\Gamma_{\mathcal{C}})}{\Gamma \vdash \mathsf{v} : t}$$

$$\text{BINARY}: \frac{\begin{array}{c}\Gamma \vdash \mathsf{le} : \tau_1 \\ \Gamma \vdash \mathsf{re} : \tau_2 \\ \Gamma \vdash \tau_1 \leq \{\ \oplus : \tau_3 \to \tau_4\ \} \\ \Gamma \vdash \tau_2 \leq \tau_3\end{array}}{\Gamma \vdash \mathsf{le} \oplus \mathsf{re} : \tau_4}$$

$$\text{UNARY}: \frac{\begin{array}{c}\Gamma \vdash \mathsf{e} : \tau_1 \\ \Gamma \vdash \tau_1 \leq \{\ \oplus : \tau_2\ \}\end{array}}{\Gamma \vdash \oplus\ \mathsf{e} : \tau_2}$$

$$\text{IF}: \frac{\begin{array}{c}\Gamma \vdash \mathsf{e}_1 : \tau_1 \\ \Gamma \vdash \tau_1 \leq Boolean \\ \Gamma \vdash \mathsf{e}_2 : \tau_2 \\ \Gamma \vdash \mathsf{e}_3 : \tau_3 \\ \Gamma \vdash \tau_2 \leq \tau_4 \\ \Gamma \vdash \tau_3 \leq \tau_4\end{array}}{\Gamma \vdash \textbf{if}\ \mathsf{e}_1\ \textbf{then}\ \mathsf{e}_2\ \textbf{else}\ \mathsf{e}_3\ \textbf{endif} : \tau_4}$$

$$\text{PROJ}: \frac{\begin{array}{c}\Gamma \vdash \mathsf{e} : \tau_1 \\ \Gamma \vdash \tau_1 \leq \{\ \mathsf{i} : \tau_2\ \}\end{array}}{\Gamma \vdash \mathsf{e.i} : \tau_2}$$

$$\text{OPER}: \frac{\begin{array}{c}\Gamma \vdash \mathsf{e} : \tau \\ \Gamma \vdash \tau \leq \{\ \mathsf{m} : (\phi_1, \ldots, \phi_n) \to \gamma\ \} \\ \Gamma \vdash \mathsf{arg_i} : \tau_i \ ^{i \in 1..n} \\ \Gamma \vdash \tau_i \leq \phi_i \ ^{i \in 1..n}\end{array}}{\Gamma \vdash \mathsf{e} \oplus \mathsf{m}(\mathsf{arg}_1, \ldots, \mathsf{arg}_n) : \gamma}$$

$$\text{LET}: \frac{\begin{array}{c}\Gamma \vdash \mathsf{e_i} : \tau_i \ ^{i \in 1..n} \\ \Gamma \vdash \tau_i \leq t_i \ ^{i \in 1..n} \\ \Gamma \setminus \{\ \mathsf{v}_i\ \} \cup \{\ \mathsf{v}_i : t_i\ \} \vdash \mathsf{be} : \tau \ ^{i \in 1..n}\end{array}}{\Gamma \vdash \mathsf{let}\ \mathsf{v}_1 : t_1 = \mathsf{e}_1, \ldots, \mathsf{v_n} : t_n = \mathsf{e_n}\ \mathsf{in}\ \mathsf{be} : \tau \ ^{i \in 1..n}}$$

$$\text{ITERATOR}: \frac{\begin{array}{c}\Gamma \vdash \mathsf{se} : \tau_1 \\ \Gamma \vdash \tau_1 \leq kind(\tau_2) \\ \Gamma \setminus \{\ \mathsf{v}\ \} \cup \{\ \mathsf{v} : \tau_2\ \} \vdash \mathsf{be} : \tau_3 \\ \Gamma \vdash kind(\tau_2) \leq \{\ iteratorOp : \tau_4 \to \tau_5\ \} \\ \Gamma \vdash \tau_3 \leq \tau_4\end{array}}{\Gamma \vdash \mathsf{se}\ \to\ iteratorOp\ (\ \mathsf{v} \mid \mathsf{be}\ ) : \tau_5}$$

$$\text{ITERATE}: \frac{\begin{array}{c}\Gamma \vdash \mathsf{se} : \tau_1 \\ \Gamma \vdash \tau_1 \leq kind(\tau_2) \\ \Gamma \vdash kind(\tau_2) \leq \{\ iterate : \tau_3 \to \tau_4\ \} \\ \Gamma \vdash \mathsf{ie} : \tau_5 \quad \Gamma \vdash \tau_5 \leq t \quad \Gamma \vdash t \leq \tau_4 \\ \Gamma \setminus \{\ \mathsf{v}, \mathsf{result}\ \} \cup \{\ \mathsf{v} : \tau_2, \mathsf{result} : t\ \} \vdash \mathsf{be} : \tau_6 \\ \Gamma \vdash \tau_6 \leq \tau_3\end{array}}{\Gamma \vdash \mathsf{se}\ \to\ iterate\ (\ \mathsf{v}; \mathsf{result} : t = \mathsf{ie} \mid \mathsf{be}\ ) : \tau_4}$$

Figure 4.5: *Type rules for other OCL expressions*

to the record type containing the field $\oplus$, and the type of right expression conforms to the parameter type of the operator, then the type of the binary expression is the return type of the operator.

The rule for the unary operator is similar to the one for the binary operators except that it takes no argument.

An If expression is composed of a *condition* expression and two branches, namely a *then* expression, and an *else* expression. The type of the *condition* expression should conform to the type *Boolean*. If we can find a common super type for the *then* and *else* types, then the type of the If expression is the common super type of its branches.

When we navigate to an attribute or an association end of a classifier in a UML model, we principally project the attribute or the association end from a record containing all the attributes and association ends of the classifier. The expression is one of the following forms:

```
object.attributeName
object.associationEndName
```

where `object` is a reference to an object that owns the attribute or association end. We also refer to `object` as a qualifier.

An operation call expression is similar to the attribute call expression except that it refers to an operation rather than an attribute. Therefore, it might have arguments. The arguments of an operation are represented as a product type $(\phi_1, \ldots, \phi_n)$. We can use either a dot ('.') or a right arrow ('→') operator to access the operation of a classifier.

A Let expression contains one or more variable declarations as depicted in the type rule. The variables declared in a Let expression are only visible in their body expression, which implies that there is no recursive declaration in the Let expression in OCL. The declared variables are added to the environment to type the body of the Let expression.

Let us compare this type rule with the type rule for the Let expression with only one variable declaration as follows:

$$\text{LET} : \frac{\begin{array}{c}\Gamma \vdash \mathsf{e} : \tau_1 \\ \Gamma \vdash \tau_1 \leq t \\ \Gamma \setminus \{\ \mathsf{v}\ \} \cup \{\ \mathsf{v} : t\ \} \vdash \mathsf{be} : \tau\end{array}}{\Gamma \vdash \mathsf{let}\ \mathsf{v} : t = \mathsf{e}\ \mathsf{in}\ \mathsf{be} : \tau}$$

Using this type rule, a Let expression with multiple declarations should be transformed into a Let expression with single declaration by introducing a nested Let expression.

For example, the Let expression

```
let v1 : t1 = e1, v2 : t2 = e2
in be
```

is transformed to

```
let v1 : t1 = e1
in let v2 : t2 = e2
   in be
```

However, in some cases the transformation results in an expression of different meaning. Consider the following example:

```
let v:Integer = 1
in let v:Integer = 2,
       w:Integer = v + 1
   in v+w
```

The scope of the variable `v` with value 1 is the body expression, which is the inner Let expression. The inner Let expression has two variable declarations: `v` with value 2 and `w` with value `v+1`. When the value `v+1` is evaluated, the value of `v` in the environment is 1. Therefore, the variable `w` evaluates to 2. In the body of the inner Let expression, the outer `v` is shadowed by the inner `v` of value 2. The body expression therefore evaluates to 4, which also becomes the value of the whole Let expression.

Now, let us now look at the case where the Let expression with multiple declarations is transformed into Let expressions with a single declaration:

```
let v:Integer = 1
in let v:Integer = 2
   in let w:Integer = v + 1
      in v+w
```

The inner `v` shadows the outer `v` in the innermost Let expression. So that, when the value `v+1` is evaluated, the value of `v` in the environment is 2 instead of 1. Therefore, the variable `w` evaluates to 3. Using the known values of `v` and `w` in the environment, the body expression of the innermost Let expression therefore evaluates to 5.

Finally, the last two rules are type rules for the loop property call expressions. An iterator expression is composed of a source expression, an iterator variable, and a body expression. The source expression of an iterator expression must have a collection type. The type of the iterator variable conforms

to the element type of the source collection. The scope of the iterator variable is its body expression. In order to type the body expression, we add the iterator variable which has type equal to the element type of the collection, to the environment. Before adding the variable to the environment, we first remove all the previous references to the variable with the same name, because they are now shadowed by the new binding. The iterator operation *iteratorOp* is a function from the body type to the return type. Therefore, the type inferred for the body expression must conform to the argument type of the iterator operation.

The type rule for the *iterate* expression is similar to the type rule for the iterator expressions, except that it requires a result variable declaration, which has a type and an initial value. The inferred type for the initial value of the result variable should conform to the declared type. Also, the declared type should conform to the result type of the *iterate* operation. The iterator variable and the result variable are added to the environment to type check the body expression.

4.6 Type Derivation

In this section we give the examples of how the derivation trees are constructed for some OCL expressions. A derivation in a given type system is a tree of judgments with leaves at the top and a root at the bottom. A valid judgment is obtained if the type rules are applied correctly.

We assume the initial environment Γ contains the following information:

$$\Gamma = (\Gamma_{\mathcal{G}}, \Gamma_{\mathcal{C}}, \Gamma_{\mathcal{L}})$$
$$\Gamma_{\mathcal{G}} = \Gamma_{uml} \cup \Gamma_{std}$$

$$
\begin{array}{lll}
\Gamma_{uml} = & \{Student \leq & \{ \quad firstName : String, \\
 & & \quad lastName : String, \\
 & & \quad age : Integer, \\
 & & \quad gender : Gender, \\
 & & \quad isActive : Boolean, \\
 & & \quad universities : Set(University), \\
 & & \quad tuitionFee : () \rightarrow Real, \\
 & & \quad birthdayHappens : () \rightarrow OclVoid \ \} \\
 & , University \leq & \{ \quad name : String, \\
 & & \quad address : String, \\
 & & \quad foundedYear : Integer, \\
 & & \quad students : Set(Student), \\
 & & \quad admitStudent : Student \rightarrow OclVoid \ \} \\
 & , MasterStudent \leq & Student \\
 & \} & \\
\end{array}
$$

$$
\begin{array}{ll}
\Gamma_{std} = & \{ \ Integer \leq Real \\
 & , \ Real \leq \{ \ > \ : Real \rightarrow Boolean \ \} \\
 & , \ Set(T) \leq Collection(T) \\
 & , \ Collection(T) \leq \{ \ forAll : Boolean \rightarrow Boolean \ \} \\
 & \}
\end{array}
$$

The Γ_{uml} is the environment which is created based on the class diagram example in Section 2.1. The Γ_{std}, on the other hand, is a subset of the subtype relations defined in the OCL 2.0 Standard Library [OMG03].

Example 1

The following invariant is a binary expression where the left expression is an operation call.

```
context Student
inv: self.tuitionFee() > 0
```

The context of this OCL expression is `Student`. Therefore, we will initialize the class and the local environment as follows:

$$
\begin{array}{ll}
\Gamma_{\mathcal{C}} = \{ \ Student \leq \{ & firstName : String, \\
 & lastName : String, \\
 & age : Integer, \\
 & gender : Gender, \\
 & isActive : Boolean, \\
 & universities : Set(University), \\
 & tuitionFee : () \ \rightarrow \ Real, \\
 & birthdayHappens : () \ \rightarrow \ OclVoid \ \} \\
\Gamma_{\mathcal{L}} = \{ \ self : Student \ \} &
\end{array}
$$

$$\frac{\textit{Premise1} \quad \dfrac{0 \in \mathbb{Z}}{\vdash \mathsf{0} : \textit{Integer}}\ \text{INT} \quad \Gamma \vdash \textit{Real} \leq \{\ >\ : \textit{Real} \rightarrow \textit{Boolean}\ \} \quad \dfrac{}{\Gamma \vdash \textit{Integer} \leq \textit{Real}}\ \text{S-INT}}{\Gamma \vdash \mathsf{self.tuitionFee}() > \mathsf{0} : \textit{Boolean}}\ \text{BINARY}$$

where

Premise1

$$\frac{\dfrac{(\mathsf{self} : \textit{Student}) \in \Gamma_{\mathcal{L}}}{\Gamma \vdash \mathsf{self} : \textit{Student}}\ \text{L-VAR} \quad \Gamma \vdash \textit{Student} \leq \{\ \textit{tuitionFee} : () \rightarrow \textit{Real}\ \}}{\Gamma \vdash \mathsf{self.tuitionFee}() : \textit{Real}}\ \text{OPER}$$

Since the example expression is a binary expression, we apply the rule BINARY at the top level. The rule BINARY has four premises.

The first premise corresponds to the left expression self.tuitionFee(). Since it is an operation call, we apply the rule OPER. To type check an operation call expression, we first have to identify the type of the qualifier. The qualifier self is a variable, hence we apply the rule L-VAR. Since the variable self is in the environment, we can obtain its type. After we have type of the qualifier, we should check whether the type conforms to the record type containing the particular operation. The environment tells us that this is the case, and we can conclude the type of the left expression.

The second premise checks the type of the right expression. Since 0 is an integer literal, we apply the rule INT.

The third premise checks whether the inferred type for the left expression *Real* conforms to the record containing the binary operator >. Our environment states that it is the case.

The last premise checks whether the inferred type for the right expression conforms to the argument type of the binary operator. Our environment states that $\textit{Integer} \leq \textit{Real}$.

Since all the premises are satisfied, we can conclude that the type of the binary expression is equal to the return type of the binary operator, which is *Boolean*.

Example 2

In this example, we construct a derivation tree for the `forAll` iterator expression:

```
context University
inv: self.students -> forAll(s | s.isActive)
```

The context of this OCL expression is `University`. Therefore, we initialize the class and the local environment as follows:

$$\Gamma_{\mathcal{C}} = \{\ University \leq \{\ \begin{array}{l} name : String, \\ address : String, \\ foundedYear : Integer, \\ students : Set(Student), \\ admitStudent : Student \rightarrow OclVoid\ \} \end{array} \}$$

$$\Gamma_{\mathcal{L}} = \{\ self : University\ \}$$

$$\frac{Premise1 \quad Premise2 \quad Premise3 \quad Premise4 \quad Premise5}{\Gamma \vdash \mathsf{self.students} \rightarrow \mathsf{forAll(s|s.isActive)} : Boolean}\ \text{ITERATOR}$$

where

Premise1

$$\frac{\dfrac{(\mathsf{self} : University) \in \Gamma_{\mathcal{L}}}{\Gamma \vdash \mathsf{self} : University}\ \text{L-VAR} \quad \Gamma \vdash University \leq \{\ students : Set(Student)\ \}}{\Gamma \vdash \mathsf{self.students} : Set(Student)}\ \text{PROJ}$$

Premise2

$$\frac{}{\Gamma \vdash Set(Student) \leq Collection(Student)}\ \text{S-SET}$$

Premise3

$$\frac{\dfrac{(\mathsf{s} : Student) \in \Gamma_{\mathcal{L}}}{\Gamma_2 \vdash \mathsf{s} : Student}\ \text{L-VAR} \quad \Gamma_2 \vdash Student \leq \{\ isActive : Boolean\ \}}{\Gamma_2 \vdash \mathsf{s.isActive} : Boolean}\ \text{PROJ}$$

Premise4

$$\Gamma \vdash Collection(Student) \leq \{\ forAll : Boolean \rightarrow Boolean\ \}$$

Premise5

$$\frac{}{\Gamma \vdash Boolean \leq Boolean}\ \text{S-REF}$$

$\Gamma_2 = \Gamma \setminus \{\ s\ \} \cup \{\ s : Student\ \}$

Since the example expression is an iterator expression, we apply the rule ITERATOR at the top level. The rule ITERATOR has five premises.

The first premise corresponds to the source expression self.students. It is a navigation to an association end of the classifier `University` so we apply the rule PROJ. In a projection, we first have to identify the type of the qualifier. The qualifier self is a variable, hence we apply the rule L-VAR, again. After we have type of the qualifier, we should check whether the type conforms to the record type containing the particular association end. The

environment tells us that this is the case, and we can conclude the type of the source expression.

The second premise checks whether the type source expression conforms to a *Collection* type. The rule S-SET in the environment matches this condition.

The third premise checks the type of the body expression `s.isActive`. To type check the body expression, we add the type of the iterator variable `s` to the environment. The type of `s` is equal to the element type of the source collection. Before adding the variable `s` to the environment, we remove the previous references to the variable with the same name. The body expression is a navigation to an attribute of the classifier, for which we apply the rule PROJ.

The fourth premise checks whether the type of the source collection conforms to the record type containing the iterator operation *forAll*.

The last premise checks whether the type of body expression conforms to the argument type of the iterator operation *forAll*. Since the types are the same, we apply the rule S-REF.

All the premises are satisfied and we may conclude the type of the iterator expression. It is equal to the return type of the iterator operation.

4.7 Type Checking OCL Expressions

While implementing the type checker, we divide the process into several phases, namely:

- processing the type environment,
- passing the type environment down the syntax tree,
- computing the type at the nodes of the tree, and
- reporting the type errors if there are inconsistencies found.

The following sections explain each phase in detail, and how they are related to each other.

4.7.1 Processing the type environment

The type environment needed for the type checking process are:

- Types from the UML model and the subtype relations between them.

- Predefined OCL types and their operations.

In order to get the UML model information, the diagram based information of the UML model should be exported into a stream based or file-based interchange format. An alternative to get the model information is through the exported XML Metadata Interchange (XMI) file [1].

The exported XMI file from the UML model is then processed into a type environment. The type environment is a Haskell datatype representing the subtype relations between the model types.

OCL predefined types and their operations should be defined in the type environment as well. The information are processed from the OCL Standard Library described in the OCL 2.0 specification [OMG03].

4.7.2 Passing down the type environment

The processed type environment from the UML model together with the type conformances defined in the OCL Standard Library forms the global environment. We use an attribute grammar (AG) and pass the type environment down the tree as an inherited attribute.

The attributes for the non terminal `Expression` is defined as follows:

```
ATTR Expression
   [   gamma  : Gamma
     |
     | tp     : Type
       errors : Errors
   ]
```

The non terminal `Expression` has three attributes: one inherited attribute and two synthesized attributes. The inherited attribute `gamma` is the type environment, while the synthesized attributes `tp` and `errors` are the computed type of an expression and the type errors respectively.

4.7.3 Computing the types

The computation of the types of the nodes of the tree is done bottom up. This process follows closely the type rules that we have defined.

[1] The XML Metadata Interchange Format (XMI) was proposed in response to an Object Management Group (OMG) Request for a Stream-based Model Interchange format. The main purpose of XMI is to enable easy interchange of data and metadata between UML modeling tools and between tools and metadata repositories in distributed heterogeneous environments [DHO01].

As the first example, let us look at the three type rules for variables from Figure 4.5.

$$\text{L-VAR} : \frac{(\mathsf{v} : t) \in \Gamma_{\mathcal{L}}}{\Gamma \vdash \mathsf{v} : t}$$

$$\text{C-VAR} : \frac{(\mathsf{v} : t) \in \Gamma_{\mathcal{C}} \quad \mathsf{v} \notin dom(\Gamma_{\mathcal{L}})}{\Gamma \vdash \mathsf{v} : t}$$

$$\text{G-VAR} : \frac{(\mathsf{v} : t) \in \Gamma_{\mathcal{G}} \quad \mathsf{v} \notin dom(\Gamma_{\mathcal{L}}) \quad \mathsf{v} \notin dom(\Gamma_{\mathcal{C}})}{\Gamma \vdash \mathsf{v} : t}$$

The implementation of those type rules are shown in the code fragment below:

```
SEM Expression
   | VariableExp
       loc.global  = globalEnvironment @lhs.gamma
          .context = head @lhs.path
       loc.(tp, errors) =
           case lookupFM (localEnvironment @lhs.gamma) @name of
             Just t  -> (t,[])
             Nothing ->lookupVar @global @context @name @einfo
       lhs.tp       = @tp
       lhs.errors   = @errors
```

In the **SEM** section, we define the semantic rules for the attributes. We refer to the `VariableExp` as a production. To refer to the attribute of a child, we write `@child.attribute`. The keyword `loc` is used to define a local attribute. We usually use a local attribute to represent a value which is used several times in the production. The scope of a local attribute is in all semantic rules of a production. The other special keyword **lhs** together with the name of the attribute is used to refer to a synthesized attribute of the nonterminal associated with a production. The complete description for the use of AG can be found in the AG System User Manual [BSL03].

The type rules for a variable expression is implemented in one production, the `VariableExp`. The code fragment also shows the look up order as denoted in the type rules. The look up to the local environment is implemented by the code fragment "`lookupFM (localEnvironment @lhs.gamma) @name`". The name of a variable is denoted by the `name`. If the variable is in the local environment, program simply returns its type, paired with an empty list of errors. The local attributes `tp` and `errors` is implemented as a pair `(tp,`

`errors`) because the attribute `errors` refers to the errors produced when the type of an expression is calculated.

If the variable is not in the local environment, we implement the code "`lookupVar @global @context @name @einfo`". The function `lookupVar` first looks up the variable in the class environment `context`. It only looks up the global environment `global` if the variable is not in the class environment.

The type rule for the If expression from Figure 4.5 is as follows:

$$\text{IF}: \frac{\begin{array}{c}\Gamma \vdash \mathsf{e}_1 : \tau_1 \\ \Gamma \vdash \tau_1 \leq \mathit{Boolean} \\ \Gamma \vdash \mathsf{e}_2 : \tau_2 \\ \Gamma \vdash \mathsf{e}_3 : \tau_3 \\ \Gamma \vdash \tau_2 \leq \tau_4 \\ \Gamma \vdash \tau_3 \leq \tau_4\end{array}}{\Gamma \vdash \textbf{if } \mathsf{e}_1 \textbf{ then } \mathsf{e}_2 \textbf{ else } \mathsf{e}_3 \textbf{ endif} : \tau_4}$$

The implementation of the above type rule is shown in the following code fragment:

```
SEM Expression
   | IfExp
     loc.global   = globalEnvironment @lhs.gamma
        .csupertp = commonSuperType @global
                            [@thenpart.tp, @elsepart.tp]
     loc.(tp, errors) =
       if subType @global @condition.tp booleanTp
       then
         if @csupertp == anyTp
         then (voidTp,[CommonTypeError
                            [@thenpart.tp,@elsepart.tp] @einfo])
         else (@csupertp, [])
       else (voidTp, [SubTypeError @condition.tp
                                   booleanTp @einfo] )
     lhs.tp       =  @tp
     lhs.errors   =  @condition.errors ++
                     @thenpart.errors ++
                     @elsepart.errors ++ @errors
```

The three sub expressions of the If expression are denoted by `condition`, `thenpart`, and `elsepart`, which correspond to the e_1, e_2, and e_3 in the type rule. The inferred type for each sub expression is accessible through the synthesized attribute `tp`. For example, to refer to the inferred type of the

`condition` expression, we write `@condition.tp`. Therefore, `@condition.tp`, `@thenpart.tp`, `@elsepart.tp` correspond to τ_1, τ_2, τ_3 in the type rule.

The code "`if subType @global @condition.tp booleanTp ...`" checks whether the inferred type for the `condition` expression conforms to the type *Boolean* as stated in the second premise in the type rule. If this premise is satisfied, then we try to find the common super type for the `thenpart` and `elsepart` types. This corresponds to the last two premises in the type rule, which implies that τ_4 is the common super type. The type of the If expression is the common super type found.

If the type conformance check fails, a subtype error will be created in `errors` attribute.

The function `commonSuperType` tries to find the common super type of two types. However, the feature of the top type *OclAny* makes the function always succeed. For example, consider the following expression

```
if true then 1 else 'hello' endif
```

The *then-part* has type *Integer*, while the *else-part* has type *String*. Both types does not conform to each other, and therefore should return a type error. Using the function *commonSuperType* to find the common super type for *Integer* and *String* will return *OclAny*, and the type error becomes undetected. To avoid this peculiarity, we have taken an anticipation by returning an error if the function `commonSuperType` ends up with *OclAny*. The code "`if @csupertp == anyTp then ..(error).. else (@csupertp, [])`" does the check. The common super type returned by the function `commonSuperType` is stored in the local attribute `csupertp`.

4.7.4 Reporting type errors

Type errors are collected during the type checking process. When the process encounters an inconsistency, a type error will be reported. The type errors are passed a synthesized attribute to the top of the tree. Besides the type information, the type error also records the sources and location where an error occurred. It is intended to report an appropriate and accurate error message. The type error message is reported in terms of the original expression.

4.7.5 The Restrictions

Because of the time limitation, there are some OCL constructs that our current implementation of the type checker can not cope with. Therefore,

the implementation poses the restrictions that the OCL constructs explained in the following sections can not be used. However, those constructs are not used often while modeling constraints in the POWER project, so that we put the implementation of the special constructs at the second priority. Also, there are already alternatives to write those constructs in their normal forms.

Shorthand for Collect

In Section 2.2.1, we show the variants of OCL expressions. One of the variants is the possibility to write the iterator expression *collect* using a shorthand notation. For example, consider the following expression.

```
self.students.firstName
```

The expression will be parsed as an attribute call expression, not as an iterator expression. Consequently, the type checker will apply the projection rule instead of the iterator rule where an error will be reported because the set of students does not have the property `firstName`.

In order to be able to type check this expression, a normalization process will be necessary to transform all the shorthands for *collect* into their normal forms.

Navigation over Associations with Multiplicity Zero or One

A navigation over associations with multiplicity zero or one results in an object at the association end. OCL allows a single object to be used as a set by treating it as a set containing one object. The usage as a set is done through an arrow operator followed by a property of the set. For example, consider the following expression.

```
context Department
inv: self.director->notEmpty()
```

The sub-expression `self.director` results in an object of type `Teacher` because the multiplicity of the association end is one. Because of the use of the arrow operator, the expression `self.director` then should have type `Set(Teacher)`.

By default, the type checker produces a type error because the object `Teacher` does not have the property `isEmpty`. This situation implies that in order to compute the type of the sub-expression, we have to know which operator applied to it. If an arrow operator is applied, a collection type is returned. If a dot operator is applied a single object type is returned.

Chapter 5

Code Generation

The last phase in our compilation process is the code generation. In the code generation process, we do not generate code for a specific programming language. Instead, we generate the RBML documents from the input OCL expressions. The main reason for this choice is that we would like to have a general system which is independent of the implementation issues.

RBML fulfills all the requirements needed for a general system in the context of the POWER project, namely *rule basedness* and *object orientedness*. Although the RBML document is not a programming language, we refer to this process as a code generation. The reason is that once we have the generated RBML document, it is possible to generate code for any target language that satisfies all the required properties.

The advantage of generating RBML document is that we can use the VALENS verification tool for verifying the document. VALENS is a knowledge verification tool developed by a third party in collaboration with the POWER project.

The formalization of how OCL constructs are translated to RBML are given in the form of translation schemes.

In the implementation of the code generator, we first define the data types which represent the RBML structures. The RBML generator reads in the augmented OCL abstract syntax tree (from the type checking process), generates the data representation for each OCL construct, and subsequently creates an RBML document from the generated data representation.

This chapter is organized as follows: in Section 5.1 we will give an overview of RBML, in Section 5.2 we present our approaches in the code generation process, and in Section 5.3 we formalize the translation schemes to RBML. Finally, in Section 5.4 we explain our implementation of the code generator.

5.1 RBML

RBML is an XML schema to represent rules in the object oriented context. It is developed by LibRT, one of the partners in the POWER project. RBML satisfies the properties required for the knowledge based components in the context of the POWER project, namely *object-orientedness* and *rule basedness*. The design and implementation of the code generator in this research project are based on the RBML v1.8 [Libb].

One of the major characteristics of RBML is that the rule is always written as a pair of *premise* and *action*. RBML can conceptually be used to describe a rule based system in which knowledge is represented as a set of *if-then* expressions (rules).

The *premise* represents a condition that evaluates to *true* or *false*.

The *action* represents the achievements if the premise is true. The action is in one of the following forms:

- *Attribute value assignment*: the assignment of a value to an attribute.
- *Instance creation*: the creation of an instance of a class.
- *Association establishment*: the establishment of an association between instances of the classes.
- *Untyped method call*: the call to a method (which may be not part of the business logic, such as: raise an exception, or print a message).

RBML contains all the concepts needed for defining a rule in the object oriented context. The root element of RBML is called *Rulebase*. The elements that form the *Rulebase* element are *Class*, *Rule*, *Association*, and *DataType*. Every element has the attributes *id* and *name* as its properties.

We will describe the syntax of those elements by using EBNF notations.

5.1.1 Class

The *Class* element contains the object oriented aspects in RBML. A class element contains the class information (*id* and *name*) and a list of properties. The properties of a class include attributes, methods, instances, and subclasses.

```
Class :=  Attribute* (PrimitiveMethod | DomainMethod)*
          Instance* SubClass*
```

5.1.2 Rule

The *Rule* element is the element in RBML to represent rule based concepts. A rule has a scope, a rule definition or a free-text rule, and zero or more bindings.

```
Rule    := Scope (RuleDef | RuleText) Binding*
RuleDef := ElseOf? Premise Action
Premise := Condition | Premise BinaryLogicalOperator?
BinaryLogicalOperator = LogicalOperator (Condition | Premise)
LogicalOperator := 'and' | 'or'
Action  := AttributeValueAssignment
         | AssociationEstablishment
         | InstanceCreation
         | UntypedMethod
Binding := ClassRef
```

There are two alternatives for defining a rule, namely:

- `RuleDef`, defines a rule by specifying the premise and action for that rule.

  The `ElseOf` is used when a rule definition refers to another rule definition. For example, the if-then-else expression results in two rules: the rule for the *then-part* and the rule for the *else-part*. In the *else-part* rule, the `ElseOf` element contains a reference to the *then-part* rule.

  The premise consists of one or more conditions separated by the logical operators.

- `RuleText`, defines a free-text rule. This alternative can be used when it is not possible to write the rule in the form of premise and action. `RuleText` makes it possible to store a rule in an unstructured format.

A binding binds a variable to a class to which the rule will apply. Both *Scope* and *Binding* elements refer to a certain class in the rule based system.

5.1.3 Association

The *Association* element can represent associations in a model, in our case associations between classes. An association has a source and destination, each having a certain multiplicity in the association. The multiplicity is in one of the following values: "1", "0..1", "0..*", or "1..*". Within an association, we can optionally specify the role name for both directions of the association.

```
Association := Source Destination Role? InverseRole?
```

The `Source` element is a reference to the source class of an association. Whereas the `Destination` element refers to the destination class of an association. The two ends of an association may have a name. `Role` name refers to the association from source to destination. While `InverseRole` name refers to the association from destination to source.

5.1.4 Data Type

The *DataType* element represents user defined datatypes, which may be the subset of the primitive types, *boolean, string, integer, real,* and *time.*

```
DataType   := Parent?  ( Constraint* | EnumValue* )
Constraint := Range+
```

There are two alternatives for defining a user defined datatype, namely:

- By specifying a constraint, which gives a range of valid values. For example, (>0...<100)
- By enumerating the values. For example, ('red', 'yellow', 'blue')

The `Parent` element allows us to define a hierarchy of the data types.

5.2 The Approaches

This section discusses our approaches in the translation from OCL to RBML. Before we discuss each approach, we will first discuss the two main approaches, namely: the *generic rule based* translation and the *conditional rule based* translation.

The *generic rule based* translation introduces a rule attribute for each OCL expression. The value of a rule attribute determines the truth value of an OCL expression. The generated rules for an OCL expression result in actions which set the rule attributes to *true* or *false* based on a set of premises. Therefore, the rule based system contains a collection of rules determining the truth value of the OCL expressions. By using this approach, we can validate a case with data by using a collection of rules, but we can not derive new data from it. The detailed explanation with examples are given in Section 5.2.2.

The *conditional rule based* translation, on the other hand, does not introduce a rule attribute for the OCL expression. From an input OCL expression, we generate rules with actions that may assign values to the attributes. Therefore, by using this approach we are able to derive new data from the existing cases given a collection of rules. However, not all OCL constructs support this translation since RBML has a restriction on the constructs that can be used. In Section 5.2.3, we show examples of constructs which are supported by this approach.

The details of our approaches in the code generation process are discussed in the following sections. In our examples we give both translations at the same time, but note that they are in no way dependent on each other. They form as it were two different programs.

5.2.1 Pattern Matching Rule

We use the pattern matching rules in our translation to RBML. The pattern matching rule iterates over many instances of a class, or over instances of several classes. When an instance meets a certain condition, the action of the rule will be executed.

A pattern matching rule consists of bindings and a rule definition. The `bind` binds a variable to a class. The pattern matching rule can have multiple bindings. The rule definition consists of an `ifmatch` clause, a `where` clause, and a `then` clause.

The `ifmatch` clause lists the binding variables that may be used in the rule, the `where` clause defines the condition that needs to be satisfied, while the `then` clause defines the action that will be executed if the condition is satisfied.

For example, let us look at the following pattern matching rule written in the Aion syntax (see Aion Manual [Ass]):

```
bind p to Person
rule "Allowance"
ifmatch p
where p.age > 18
then p.allowance := 800
end
```

This rule applies to every instances of the class *Person*, and sets the value of the attribute *allowance* to 800, if the *age* is greater than 18.

Since a pattern matching rule iterates over instances of a class, it makes it possible for us to translate OCL iterator expressions using the pattern matching rules.

For OCL iterator expressions, the binding will be a collection of instances to which the expressions belong. For non-iterator expression, the binding will be the context in which the expression is defined.

The corresponding RBML rule for the above syntax is as follows:

```
Rule {name = "Allowance" }
   RuleDef
      Premise : p.age > 18
      Action  : p.allowance := 800
   Binding {name = "p" }
      Class {idref = "Person"}
```

The `where` clause becomes a *Premise*, the `then` clause becomes an *Action*, while the `bind` clause becomes *Binding*. Both the *Rule* and *Binding* element in RBML have an attribute *name* to store the name of the rule and the binding respectively. The attribute *idref* under the *Binding* refers to the class to which the binding variable becomes bound.

Notice the reordering of information. This will also be the case in the following examples.

5.2.2 Generic Rule Based Translation

In the *generic rule based* translation, we generate a unique attribute R_n to represent the value of each rule. Those attributes belong to a generic class *RBMLBase* and have initial value *unknown*. This approach will generate two rules, namely:

first rule: the *premise* is the original OCL expression, and the *action* is an assignment of *true* to the rule attribute.

second rule: the *premise* is the negation of the original OCL expression, and the *action* is an assignment of *false* to the rule attribute.

For example, suppose we have the following OCL expression:

```
context Person
inv: self.married implies self.age > 18
```

The translation of the OCL invariant to RBML is then as follows:

```
Rule { id= 1, name = R1.1 }
   Scope { idref = Person }
   RuleDef
      Premise : not (person.married) or (person.age > 18)
      Action : R1 := true
   Binding { id= 2, name = person }
      Class { idref = Person }
;Rule { id= 3, name = R1.2 }
   Scope { idref = Person }
   RuleDef
      Premise : (person.married) and not (person.age > 18)
      Action : R1 := false
   Binding { id= 4, name = person }
      Class { idref = Person }
```

In the above translation, we generate a unique *id* and *name* for each rule. The scope of this rule is the context to which the OCL expression belongs.

In the first rule, the premise is the original OCL expression which is translated to its logical equivalence, namely: E_1 `implies` $E_2 \equiv$ `not` E_1 `or` E_2. The action is the assignment of *true* to the rule attribute R_1. R_1 is the meta attribute which is generated by the system to hold the truth value of the OCL expression. It is not part of the OCL expression.

In the second rule, the premise is the negation of the original expression with the translation to its logical equivalence, namely: `not` (E_1 `or` E_2) $\equiv$ (E_1 `and not` E_2). The action is the assignment of *false* to the rule attribute R_1.

Since we use the pattern matching rule approach in our translation, as discussed in Section 5.2.1, we need to create bindings on which a certain rule will be executed. In most OCL expressions, the binding variable will be bound to the context where the OCL expressions are introduced. All the attributes in the expression are qualified by their binding variables. We use binding variables which have the same name as the class name, but starting with a lower case letter.

In our example, we replace all the occurrences of `self` with `person`, because the attributes are now bound to this binding variable. Therefore, we now write `person.married` and `person.age`.

5.2.3 Conditional Rule Based Translation

In the *conditional rule based* translation, we first check whether a particular OCL expression can be written in the form of a pair of *premise* and *action*. The translation will only be done if it is possible.

Using this approach, the implication expression "E_1 `implies` E_2" will be translated into two if-then expressions which are logically equivalent, namely:

`if` E_1 `then` E_2
`if not` E_2 `then not` E_1

The if-then expressions can be easily written as a pair of premise and action in RBML, where the *if-part* becomes a premise, and the *then-part* becomes an action of the rule definition.

This approach has some restrictions: the expression which acts as an action, namely E_2 and `not` E_1 should satisfy the requirements for an action in RBML as discussed in Section 5.1. The requirements state that the expression should be able to be written as an *attribute value assignment*, an *instance creation*, an *association establishment*, or an *untyped method*.

Our example from the previous section is translated into two if-then expressions, namely:

```
if self.married then self.age > 18                    (1)
if not (self.age > 18) then not (self.married)        (2)
```

However, the translations to RBML for these expressions are not supported since the then-part of the expression (1) can not be written in form of *action* in RBML.

Table 5.1 shows several possible forms of the implication expressions and how they are translated to RBML by using the *conditional rule based* approach. The translation results in two rules which, together, represent the same logic as the initial OCL expressions. If one of the translations is not supported, then the other rule will not be generated either. As we can see from the examples, the actions of the rules assign values to the attributes. Given these rules and a set of cases, we are able to derive values for the attributes.

The fact that we assign a value to an attribute does not mean that we change the value of the attribute since the value assigned the attribute is derived from the constraint itself: an equality constraint or a boolean attribute. In the case of a boolean attribute, we derive the value `true` or `false` depending on the constraint.

We summarize the expressions that are supported in translation of logical implication expression E_1 `implies` E_2 by using the *conditional rule based* translation:

The premise of the implication E_1 should be in one of the following forms:

E_1 `implies` E_2	`if` E_1 `then` E_2	`if not` E_2 `then not` E_1
x `implies` y	*Premise:* x *Action:* y := true	*Premise:* not y *Action:* x := false
x `implies` y>0	*Premise:* x *Action:*(not supported)	*Premise:* not (y>0) *Action:* x := false
x=18 `implies` y=18	*Premise:* x = 18 *Action:* y := 18	*Premise:* not (y=18) *Action:*(not supported)
x<>18 `implies` y=18	*Premise:* x <> 18 *Action:* y := 18	*Premise:* not (y=18) *Action:* x := 18
x=true `implies` y=false	*Premise:* x = true *Action:* y := false	*Premise:* not (y=false) *Action:* x := false

Table 5.1: *Some examples translation implication expression to RBML*

- Boolean attribute, e.g. "isMarried", "isEmployed".
- Boolean equality expression, e.g. "x = true".
- Inequality expression, e.g. "x <> 0".

The action of the implication E_2 should be in one of the following forms:

- Boolean attribute.
- Equality expression.

As a consequence of this approach, the generated rules are not complete because of the limited number of supported constructs.

5.3 Translation Schemes

A translation scheme defines how a certain language construct is translated to the target language by writing the target languages as a function of the language construct [DS03].

$$
\begin{aligned}
&Scheme[[Construct]](p_1, \ldots, p_n) \\
&\equiv\ TargetLanguagePart_1; \\
&\quad TargetLanguagePart_2; \\
&\quad \ldots \\
&\quad TargetLanguagePart_n;
\end{aligned}
$$

The scheme is applied to an OCL construct to produce a sequence of target language constructs which are separated by semicolons. The extra information needed for the translation is given in form of parameters (p_1,...,p_n).

We might need many different schemes depending on the language construct. The translation scheme for the OCL expression is called $\mathcal{E}$. Although we use the concrete syntax in the argument of a scheme, the translation scheme follows the structure of the abstract syntax.

Besides the translation schemes for the OCL expressions, we also define a translation scheme for the attribute value assignment, called $\mathcal{A}$. We distinguish the translation schemes because an attribute value assignment is not an OCL expression.

To increase the readability, the output of the translation is written in a tree like document rather than in the form of the XML document. However, the actual output will be an XML document.

The translation scheme for the attribute value assignment is as follows:

$\mathcal{A}$[[$v := e$]] env
$\equiv$ `Action`
`AttributeValueAssignment`
`AttributeReference`
`Attribute` { `idref` = $env[v]_{id}$ }
`TypedExpression`
$\mathcal{E}$[[e]]

An attribute value assignment results in action of a rule based system. The attribute reference is a reference to the attribute of a particular instance of a class within the system. For this translation, we need an additional parameter *env*, which is the environment containing the class information and its properties. $env[v]_{id}$ extracts the *id* for v out of the environment *env*.

The corresponding RBML document for the above translation will be:

```
<Action>
    <AttributeValueAssignment>
        <AttributeReference>
            <Attribute  idref = "..."/>
            <TypedExpression >
                ...
                ...
            </TypedExpression >
        </AttributeReference>
    </AttributeValueAssignment>
</Action>
```

5.3.1 Literal

RBML defines a number of predefined literals, such as *true*, *false*, *null*, *unknown*, etc. The list of predefined literals is passed as the parameter *predefLit*. If a literal is an element of the predefined literal, then the literal value is saved to the attribute *predefined*, and to the attribute *freeformat* otherwise.

The translation scheme for the literal expression is as follows:

$\mathcal{E}$[[*lit*]] *predefLit*
≡ `TypedExpression`
  if *lit* elem *predefLit* then
      `Literal` { `freeformat=""`, `predefined=`*lit* }
  else
      `Literal` { `freeformat=`*lit*, `predefined=""` }

5.3.2 Binary Expression

We distinguish several translation schemes for the binary expression "$e_1 \oplus e_2$". Each scheme corresponds to a set of binary operators as follows:

- *Arithmetic Expression* corresponds to the operators { '+', '-', '*', '/' }.
- *Relational Expression* corresponds to the operators { '<', '>', '<=', '>=', '=', '<>', 'and', 'or' }.
- *Implication Expression* corresponds to the operator { 'implies' }.

Arithmetic Expression

The translation scheme for the arithmetic expression is as follows:

$\mathcal{E}$[[*le* $\oplus$ *re*]]
≡ `ArithmeticExpression`
      `LeftHand`
        $\mathcal{E}$[[*le*]]
      `ArithmeticOperator` { `operator` = $\mathcal{O}$[[$\oplus$]] }
      `RightHand`
        $\mathcal{E}$[[*re*]]

where $\oplus$ ∈ { '+', '-', '*', '/' }

The binary operators introduce a new translation scheme, namely $\mathcal{O}$. For each binary operator in OCL, there is a corresponding operator in RBML.

$\mathcal{O}$[[+]]
≡ `plus`

$\mathcal{O}$[[−]]
≡ `minus`

$\mathcal{O}$[[∗]]
≡ `times`

$\mathcal{O}$[[/]]
≡ `divided-by`

Relational Expression

The relational expression is a binary expression where its operator is the element of { '<', '>', '<=', '>=', '=', '<>', 'and', 'or' }. We can only perform the generic rule based translation for these expressions, since they do not satisfy the requirements for a conditional rule based translation, except for the equality expression.

The translation scheme for the relational expression is as follows:

$\mathcal{E}$[[*le* ⊕ *re*]] (*env*, *ctx*)
≡ `Rule` { `id` = *ctr*, `name` = R_n.1 }
  `RuleDef`
    $\mathcal{E}$[[(*le* ⊕ *re*) `and` *cond*]]
    $\mathcal{A}$[[R_n := *true*]]
  $\mathcal{B}$[[*ctx*]] (*ctr*, *env*)
;`Rule` { `id` = *ctr*, `name` = R_n.2 }
  `RuleDef`
    $\mathcal{E}$[[*not* (*le* ⊕ *re*)]]
    $\mathcal{A}$[[R_n := *false*]]
  $\mathcal{B}$[[*ctx*]] (*ctr*, *env*)

if (⊕=="=") and (*isAttribute* le) then
;`Rule` { `id` = *ctr*, `name` = R_n.3 }
  `RuleDef`
    $\mathcal{E}$[[*true*]]
    $\mathcal{A}$[[*le* := *re*]]
  $\mathcal{B}$[[*ctx*]] (*ctr*, *env*)
endif
if (⊕=="=") and (*isAttribute* re) then
;`Rule` { `id` = *ctr*, `name` = R_n.4 }
  `RuleDef`

$$
\begin{array}{l}
\quad\quad \mathcal{E}[[true]] \\
\quad\quad \mathcal{A}[[re := le]] \\
\quad\quad \mathcal{B}[[\ ctx\ ]]\ (ctr, env) \\
\text{endif}
\end{array}
$$

where $\oplus \in$ { '<', '>', '<=', '>=', '=', '<>', 'and', 'or' }

$cond = (R_n = \text{true or } R_n = \text{unknown})$

$ctr = freshId()$

$n = freshRuleNumber()$

$isAttribute\ (\texttt{AttributeCall}\ \_\ \_\ \_\ \_) = \texttt{True}$

$isAttribute\ \_ = \texttt{False}$

The first two rules in the scheme correspond to the *generic rule based* translation approach. For each expression where this approach is applied, we generate a rule attribute R_n. This attribute belongs to the generic class *RBMLBase*, as described in Section 5.2.2. Since we use the pattern matching rule, we add an additional condition (R_n = true or R_n = unknown) to ensure that the attribute R_n is only set to `true` if the premise is satisfied by all the instances of the class where the rule applied. Therefore, our example in Section 5.2.2 is not complete yet. We should add this additional condition to the premise in order for the rule to be correct.

The last two rules in the scheme correspond to the *conditional rule based* translation approach. This approach can only be applied to the equality expression where it requires that the expression is an attribute call expression. The function *isAttribute* does the check.

In above translation scheme, we also introduce a translation scheme for binding, which is called $\mathcal{B}$, as follows:

$$
\begin{array}{l}
\mathcal{B}[[\ nm\ ]]\ (i, env) \\
\equiv\ \texttt{Binding}\ \{\ \texttt{id} = i,\ \texttt{name} = nm\ \} \\
\quad\quad \texttt{Class}\ \{\ \texttt{idref} = env[nm]_{id}\ \}
\end{array}
$$

We use the helper functions *freshId* and *freshRuleNumber* to return a fresh *id* and a fresh rule number respectively. The function *freshId* is called every time we want to assign a fresh *id* to an element. Whereas the function *freshRuleNumber* is called only once for each OCL expression.

Example Translation

The examples of the translation in this section and the following sections are written in the pattern matching rule syntax. The main reason is to make it easier to understand. In addition, the pattern matching rule syntax has a straightforward translation to RBML syntax as described in Section 5.2.1.

```
context University
inv: self.yearFounded = 1900
```

The translation of the above example to RBML produces three rules. The generated rules $\texttt{R}_n\texttt{.1}$ and $\texttt{R}_n\texttt{.2}$ correspond to the first two rules in the translation scheme for the relational expression. Since the binary operator and the left expression match the condition in the scheme, we also generate the third rule.

In the generated rules, the qualifier *self* is replaced with the binding variable *university*.

$\texttt{R}_n\texttt{.1}$ `bind` $university$ `to` $University$
`ifmatch` $university$
`where` $(university.yearFounded = 1900)$
`and` $(R_n = \texttt{true}$ `or` $R_n = \texttt{unknown})$
`then` $(R_n := \texttt{true})$
`end`

$\texttt{R}_n\texttt{.2}$ `bind` $university$ `to` $University$
`ifmatch` $university$
`where` $(\texttt{not}\ university.yearFounded = 1900)$
`then` $(R_n := \texttt{false})$
`end`

$\texttt{R}_n\texttt{.3}$ `bind` $university$ `to` $University$
`ifmatch` $university$
`where true`
`then` $(university.yearFounded := 1900)$
`end`

Implication Expression

The translation scheme for the implication expression is as follows:

$\mathcal{E}[\![\ le$ `implies` $re\ ]\!]\ (env, ctx)$
$\equiv$ `Rule {` `id` $= ctr,$ `name` $= R_n.1$ `}`
`RuleDef`
$\mathcal{E}[\![$ `not` le `or` re `and` $cond\ ]\!]$
$\mathcal{A}[[R_n = true]]$
$\mathcal{B}[\![\ ctx\ ]\!]\ (ctr, env)$
`;Rule {` `id` $= ctr,$ `name` $= R_n.2$ `}`
`RuleDef`
$\mathcal{E}[\![\ le$ `and (` `not` re `) ]\!]`
$\mathcal{A}[[R_n := false]]$

$\mathcal{B}$[[*ctx*]] (*ctr*, *env*)

if (*isPremise* le) and (*isAction* re) then
;`Rule` { `id` = *ctr*, `name` = R_n.3 }
`RuleDef`
$\mathcal{E}$[[*le*]]
$\mathcal{A}$[[*re* := *true*]]
$\mathcal{B}$[[*ctx*]] (*ctr*, *env*)
;`Rule` { `id` = *ctr*, `name` = R_n.4 }
`RuleDef`
$\mathcal{E}$[[`not` *re*]]
$\mathcal{A}$[[*le* := *false*]]
$\mathcal{B}$[[*ctx*]] (*ctr*, *env*)
endif

where *cond* = (R_n = true or R_n = unknown)
ctr = *freshId*()
n = *freshRuleNumber*()
isPremise (`BinaryExp _ "<>" _`)= `True`
isPremise (`BinaryExp _ "=" (BooleanLiteral _)`)= `True`
isPremise (`BooleanLiteral _`) = `True`
isPremise _ = `False`

isAction (`BinaryExp _ "=" _`) = `True`
isAction (`BooleanLiteral _ `) = `True`
isAction _ = `False`

The first rule translates the implication expression to its logical equivalence, while the second rule translates to the negation of the expression. The last two rules require that both the left and right expressions fulfill the conditions for the translation of the implication expression to RBML, as discussed in the section 5.2.3.

Example Translation

```
context Student
inv: self.course.notEmpty() implies self.isActive
```

R_n.1 `bind` *student* `to` *Student*
`ifmatch` *student*
`where` (`not` *student.course.notEmpty*() `or` *student.isActive*)
`and` (R_n = `true` `or` R_n = `unknown`)
`then` (R_n := `true`)
`end`

```
R_n.2   bind student to Student
        ifmatch student
        where (student.course.notEmpty() and not student.isActive
        then (R_n := false)
        end
R_n.3   bind student to Student
        ifmatch student
        where student.course.notEmpty()
        then (student.isActive := true)
        end
R_n.4   bind student to Student
        ifmatch student
        where not student.isActive
        then (student.course.notEmpty() := false)
        end
```

5.3.3 Unary Expression

The translation scheme for the OCL unary expression is the simple one since the unary expression is basically a negation of an expression. The premise and condition element in RBML include the attribute *not*. This attribute has a default value *false*. When we want to create a negation of a premise or a condition, we simply set the attribute *not* to *true*.

$\mathcal{E}$[[$\oplus$ e]]
$\equiv$ $\mathcal{E}$[[e]] *isnot*

where: *isnot* = `true`
$\oplus$ $\in$ { '-', 'not' }

The parameter *isnot* carries the value of the attribute *not*. Translating an unary expression is equal to translating the expression, and then set the attribute *not* according to the value of the parameter *isnot*.

5.3.4 If Expression

The OCL If expression is typical for the translation to RBML since all other expressions are translated to a kind of if-then form in RBML. The translation scheme for the If expression is as follows:

$\mathcal{E}$[[`if` ce `then` te `else` ee `endif`]] (env, ctx)
$\equiv$ `Rule` { `id` = ctr, `name` = R_n.1 }

```
        RuleDef
            E[[ce]]
            A[[R_n := R_te]]
        B[[ ctx ]] (ctr, env)
    ;Rule { id = ctr, name = R_n.2 }
        RuleDef
            E[[ not ce ]]
            A[[R_n := R_ee]]
        B[[ ctx ]] (ctr, env)

    if (isAction te) and (isAction ee) then
    ;Rule { id = ctr, name = R_n.3 }
        RuleDef
            E[[ ce ]]
            A[[ te ]]
        B[[ ctx ]] (ctr, env)
    ;Rule { id = ctr, name = R_n.4 }
        RuleDef
            ElseOf { idref = env[R_n.3]_id }
            E[[ true ]]
            A[[ ee ]]
        B[[ ctx ]] (ctr, env)
    endif

  where ctr = freshId()
        n = freshRuleNumber()
```

We perform both generic and conditional rule based translations for the If expression. The conditional rule based translation requires that both the *then* and *else* expressions can be translated to the *action* in RBML. The function *isAction* does the checks. The `ElseOf` element in the last rule refers to the previous rule as shown in its id reference *idref*. This element denotes that the rule represents an *else* part of another rule.

Example Translation

```
context Course
inv: if courseType = CourseType :: preliminary
     then self.credit = 5
     else self.credit = 6
     endif

R_n.1   bind course to Course
        ifmatch course
```

```
        where (course.courseType = CourseType :: preliminary)
        then (R_n := R_(credit=5))
        end
R_n.2   bind course to Course
        ifmatch course
        where (not course.courseType = CourseType :: preliminary)
        then (R_n := R_(credit=6))
        end
R_n.3   bind course to Course
        ifmatch course
        where (course.courseType = CourseType :: preliminary)
        then (credit := 5)
        end
R_n.4   bind course to Course
        elseof R_n.3
        ifmatch course
        where true
        then (credit := 6)
        end
```

5.3.5 Attribute Call Expression

The attribute call expression has a straightforward translation in RBML. It is a reference to the attribute of the instance of a class identified by the qualifier *se*. If the qualifier is empty, then this expression refers to the attribute of the object in the current scope.

$\mathcal{E}$[[*se.attrname*]] *env*
≡ `AttributeReference`
`Qualifier` $\mathcal{E}$[[*se*]]
`Attribute` { `idref` = $env[attrname]_{id}$ }

5.3.6 Operation Call Expression

The operation call expression will be translated to the *TypedMethod* in RBML. It is a reference to the method of the instance of a class identified by the qualifier *se*. The passed arguments should match the arguments of the referred method. The translation scheme is as follows:

$\mathcal{E}$[[$se \rightarrow op\ (arg_1, \ldots, arg_n)$]] *env*
≡ `TypedMethod`
`CalledMethod`

$$
\begin{array}{l}
\quad\quad\quad\quad\texttt{Qualifier}\ \mathcal{E}[[se]] \\
\quad\quad\quad\quad\texttt{Method}\ \{\ \texttt{idref} = env[op]_{id}\ \} \\
\quad\quad\quad \mathcal{P}[[arg_1]]\ (env, op, 1), \ldots, \mathcal{P}[[arg_n]]\ (env, op, n)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{P}[[\ arg\ ]]\ (env, op, n) \\
\equiv\ \texttt{PassedArgument} \\
\quad\quad\quad\texttt{Argument}\ \{\ \texttt{idref} = env[\ arg(op, n)\ ]_{id}\ \} \\
\quad\quad\quad\texttt{ArgumentExpression}\ \mathcal{E}[[\ arg\ ]]
\end{array}
$$

The translation scheme for the parameter of an operation is called $\mathcal{P}$. It expects three parameters: the class information *env*, the operation name *op*, and the n^{th} argument of the operation. The helper function *arg(op,n)* returns the name of the n^{th} argument of an operation.

5.3.7 Loop Expression

The loop expressions have the most complicated translations. For each iterator method, we have different translation scheme.

forAll

We generate four rules in the translation of the *forAll* iterator expression into RBML.

In addition to the binding to a context, written as $\mathcal{B}[[\ ctx\ ]]$, two new bindings are introduced: a binding to the element of the source collection $\mathcal{B}[[\ elem\ ]]$, and a binding to the source collection $\mathcal{B}[[\ sourceColl_n\ ]]$. Since a loop expression works on a collection, these bindings are necessary to enable us to access the properties of an element of the collection, to relate the element to the collection, and to relate the context to the collection.

The translation scheme is as follows:

$$
\begin{array}{l}
\mathcal{E}[[\ se\ \rightarrow\ forAll\ (\ be\ )\ ]]\ (env, ctx) \\
\equiv\ \texttt{Rule}\ \{\ \texttt{id} = ctr,\ \texttt{name} = R_n.1\ \} \\
\quad\quad\quad\texttt{RuleDef} \\
\quad\quad\quad\quad\mathcal{E}[[\ cond_2\ \texttt{and}\ cond_3]] \\
\quad\quad\quad\quad\mathcal{A}[[R_n := \texttt{true}]] \\
\quad\quad\quad\mathcal{B}[[\ ctx\ ]]\ (ctr, env) \\
\quad ;\texttt{Rule}\ \{\ \texttt{id} = ctr,\ \texttt{name} = R_n.2\ \} \\
\quad\quad\quad\texttt{RuleDef} \\
\quad\quad\quad\quad\mathcal{E}[[\ ctxforAll = \texttt{false}\ ]] \\
\quad\quad\quad\quad\mathcal{A}[[\ R_n := \texttt{false}\ ]]
\end{array}
$$

$\mathcal{B}$[[*ctx*]] (*ctr*, *env*)
;`Rule` { `id` = *ctr*, `name` = R_n.3 }
`RuleDef`
$\mathcal{E}$[[*elem.be* `and` $cond_1$ `and` $cond_4$]]
$\mathcal{A}$[[$sourceColl_n.forAll$:= `true`]]
$\mathcal{B}$[[*elem*]] (*ctr*, *env*)
$\mathcal{B}$[[$SourceColl_n$]] (*ctr*, *env*)
$\mathcal{B}$[[*ctx*]] (*ctr*, *env*)
;`Rule` { `id` = *ctr*, `name` = R_n.4 }
`RuleDef`
$\mathcal{E}$[[`not` (*elem.be*) `and` $cond_1$]]
$\mathcal{A}$[[$sourceColl_n.forAll$:= `false`]]
$\mathcal{B}$[[*elem*]] (*ctr*, *env*)
$\mathcal{B}$[[$SourceColl_n$]] (*ctr*, *env*)
$\mathcal{B}$[[*ctx*]] (*ctr*, *env*)

where $ctxforAll = ctx.sourceColl_n.forAll$
$elem = ctx.sourceColl_n.elem$
$cond_1 = (sourceColl_n$ `includes` $elem)$
`and` $(sourceColl_n = ctx.sourceColl_n)$
$cond_2 = (ctxforAll =$ `true` `or` $ctxforAll =$ `unknown`)
$cond_3 = (R_n =$ `true` `or` $R_n =$ `unknown`)
$cond_4 = (sourceColl_n.forAll =$ `true`
`or` $sourceColl_n.forAll =$ `unknown`)
ctr = *freshId*()
n = *freshRuleNumber*()

The translation scheme above can be described as follows:

To represent the source collection of an iterator expression, a system generated class $sourceColl_n$ is generated for each iterator expression. The convention for naming the system generated collection class is:

```
element_type_name + "Coll" ++ counter
```

For example, $StudentColl_n$ is to name a collection of elements of type *Student* in the n^{th} iterator expression. This class has an attribute to store the evaluation value for every instance of the class. This attribute has the same name with the name of the iterator operation. For example, if we operate on the iterator operation *forAll*, the attribute is named *forAll*.

In the scope of an element of the source collection, the rules R_n.3 and R_n.4 are evaluated on every instance of the class *elem* which is associated to the source collection. The result of the evaluation is stored in the attribute *forAll* of the class $sourceColl_n$.

The additional condition $sourceColl_n$ `includes` *elem* `and` ($sourceColl_n$ = $ctx.sourceColl_n$) is to prevent the expression to be evaluated on every instances of the class *elem*. Instead, it is only evaluated on the instances of *elem* which are associated to the $sourceColl_n$.

The condition ($sourceColl_n.forAll$ = `true or` $sourceColl_n.forAll$ = `unknown`) is to prevent the attribute *forAll* is set to *true* if at least one of the instances of the class *elem* does not match the condition *be*.

In the scope of the context *ctx*, the rules R_n.1 and R_n.2 are evaluated on every instance of the class $sourceColl_n$ which is associated to the class *ctx*. The result of the evaluation determines the truth value of the expression, represented by the attribute R_n.

The condition ($forall$= `true or` $forall$= `unknown`) is to prevent the rule is set to *true* if the attribute *forAll* of one of the instances of the class $sourceColl_n$ has value *false*. Similary for the second condition (R_n = `true or` R_n = `unknown`), if the rule has been set to *false*, there is no way for it to be set to *true* again.

In the rules R_n.3 and R_n.4, we create three bindings of different types:

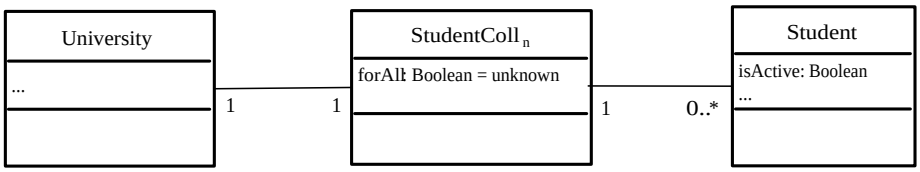
- binding for the element of the source collection (*elem*),
- binding for the source collection of the iterator expression ($SourceColl_n$), and
- binding for the current context (*ctx*).

The name of the binding is equal to the name of the referred class, started with a lower case letter.

Example Translation

```
context University
inv: self.students -> forAll(isActive)
```

The source expression `self.students` evaluates to a set of students. Therefore, we create a class to represent the source collection in the iterator expression, called $\texttt{StudentColl}_n$. The class $\texttt{StudentColl}_n$ has one attribute: the attribute `forAll`. The attribute is of type *Boolean*, and has an initial value `unknown`. The class $\texttt{StudentColl}_n$ is associated to the class `University` and the class `Student` as shown below.



The name of each association end is the same as the name of the class at the association end, starting with a lower case letter.

The translation to RBML is as follows:

```
R_n.1   bind university to University
        ifmatch university
        where (university.studentColl_n.forall = true
               or university.studentColl_n.forall = unknown)
               and (R_n = true or R_n = unknown)
        then (R_n := true)
        end
R_n.2   bind university to University
        ifmatch university
        where (university.studentColl_n.forall = false)
        then (R_n := false)
        end
R_n.3   bind student to Student
        bind studentColl_n to StudentColl_n
        bind university to University
        ifmatch student, studentColl_n, university
        where student.isActive
               and (studentColl_n includes student)
               and (studentColl_n = university.studentColl_n)
               and (studentColl_n.forAll = true or
                    studentColl_n.forAll = unknown)
        then (studentColl_n.forAll := true)
        end
R_n.4   bind student to Student
        bind studentColl_n to StudentColl_n
        bind university to University
        ifmatch student, studentColl_n, university
        where (not student.isActive)
               and (studentColl_n includes student)
               and (studentColl_n = university.studentColl_n)
        then (studentColl_n.forAll := false)
        end
```

exists

The translation scheme for the *exists* iterator expression is similar with the one for the *forAll*, except for the additional conditions to be checked. In the *exists* operation, the rule will be true if one of the instances of the class

satisfies the condition. In the *forAll* operation, the rule will only be true if all instances of the class satisfy the condition.

$\mathcal{E}$[[*se* $\rightarrow$ *exists* (*be*)]] (*env*, *ctx*)
$\equiv$ `Rule` { `id` = *ctr*, `name` = R_n.1 }
`RuleDef`
$\mathcal{E}$[[(*ctxexists* = `true` `or` *ctxexists* = `unknown`)]]
$\mathcal{A}$[[R_n := `true`]]
$\mathcal{B}$[[*ctx*]] (*ctr*, *env*)
;`Rule` { `id` = *ctr*, `name` = R_n.2 }
`RuleDef`
$\mathcal{E}$[[*ctxexists* = `false` `and` (R_n = `unknown`)]]
$\mathcal{A}$[[R_n := `false`]]
$\mathcal{B}$[[*ctx*]] (*ctr*, *env*)
;`Rule` { `id` = *ctr*, `name` = R_n.3 }
`RuleDef`
$\mathcal{E}$[[*elem.be* `and` *cond*]]
$\mathcal{A}$[[$sourceColl_n.exists$:= `true`]]
$\mathcal{B}$[[*elem*]] (*ctr*, *env*)
$\mathcal{B}$[[$SourceColl_n$]] (*ctr*, *env*)
$\mathcal{B}$[[*ctx*]](*ctr*, *env*)
;`Rule` { `id` = *ctr*, `name` = R_n.4 }
`RuleDef`
$\mathcal{E}$[[`not` (*elem.be*) `and` *cond* ($sourceColl_n.exists$ = `unknown`)]]
$\mathcal{A}$[[$sourceColl_n.exists$:= `false`]]
$\mathcal{B}$[[*elem*]](*ctr*, *env*)
$\mathcal{B}$[[$SourceColl_n$]](*ctr*, *env*)
$\mathcal{B}$[[*ctx*]] (*ctr*, *env*)

where *cond* = (*sourceColl*$_n$ `includes` *elem*)
`and` ($sourceColl_n$ = *ctx*.$sourceColl_n$)
ctxexists = *ctx*.$sourceColl_n$.*exists*
ctr = *freshId*()
n = *freshRuleNumber*()

Example Translation

```
context University
inv: self.students -> exists(firstName = 'Eelco')
```

R_n.1 `bind` *university* `to` *University*
`ifmatch` *university*
`where` (*university*.$studentColl_n$.*exists* = `true`

```
                or university.studentColl_n.exists = unknown)
        then (R_n := true)
R_n.2   bind university to University
        ifmatch university
        where (university.studentColl_n.exists = false)
                and (R_n = unknown)
        then (R_n := false)
R_n.3   bind student to Student
        bind studentColl_n to StudentColl_n
        bind university to University
        ifmatch student, studentColl_n, university
        where (student.firstName = 'Eelco')
                and (studentColl_n includes student)
                and (studentColl_n = university.studentColl_n)
        then (studentColl_n.exists := true)
R_n.4   bind student to Student
        bind studentColl_n to StudentColl_n
        bind university to University
        ifmatch student, studentColl_n, university
        where not (student.firstName = 'Eelco')
                and (studentColl_n includes student)
                and (studentColl_n = university.studentColl_n)
                and (studentColl_n.exists = unknown)
        then (studentColl_n.exists := false)
```

select

The *select* iterator expression itself is not an invariant since it returns a collection, not a boolean value. Consequently, we can not perform the *generic rule based* translation on this expression. However, the resulting collection can be used by another expression to create an invariant.

$\mathcal{E}$[[*se* → *select* (*be*)]] (*env*, *ctx*)
≡ Rule { id = *ctr*, name = R_n.1 }
RuleDef
$\mathcal{E}$[[*elem.be* and *cond*]]
$\mathcal{A}$[[*sourceColl*$_n$.*select* := including(*sourceColl*$_n$.*select*, *elem*)]]
$\mathcal{B}$[[*elem*]](*ctr*, *env*)
$\mathcal{B}$[[*SourceColl*$_n$]] (*ctr*, *env*)
$\mathcal{B}$[[*ctx*]](*ctr*, *env*)

where *cond* = (*sourceColl*$_n$ includes *elem*)
and (*sourceColl*$_n$ = *ctx.sourceColl*$_n$)

$ctr = freshId()$
$n = freshRuleNumber()$

We use the method `including` to include an element to a collection. This method belongs to the *Domain* class (see Section 5.4.1).

Example Translation

The following example shows the use of the *select* iterator expression within an operation call expression *size* and how they are translated to RBML.

```
context University
inv: self.students -> select(lastName = 'Visser')->size() > 0
```

```
R_n.1   bind student to Student
        bind studentColl_n to StudentColl_n
        bind university to University
        ifmatch student, studentColl_n, university
        where (student.lastName = 'Visser')
                and (studentColl_n includes student)
                and (studentColl_n = university.studentColl_n)
        then studentColl_n.select := including (studentColl_n.select, student)
        end
R_n.2   bind studentColl_n to StudentColl_n
        bind university to University
        ifmatch studentColl_n, university
        where studentColl_n_select.size() > 0
                and (R_n = true or R_n = unknown)
        then (R_n := true)
        end
R_n.3   bind studentColl_n to StudentColl_n
        bind university to University
        ifmatch studentColl_n, university
        where (not studentColl_n_select.size() > 0)
        then (R_n := false)
        end
```

The rule `R`$_n$`.1` in the above example implements the translation scheme for the *select* operation. This rule results in an intermediate collection value that are used by the other rules: `R`$_n$`.2` and `R`$_n$`.3`. The rules `R`$_n$`.2` and `R`$_n$`.3` implement the translation scheme for the relational expression (see Section 5.3.2). We use a canonical name `studentColl`$_n$`_select` to refer to the intermediate collection.

reject

The translation of the *reject* iterator expression is similar with the one for the *select* iterator expression, except that it rejects the instances that satisfy the condition.

$\mathcal{E}$[[*se* $\rightarrow$ *reject* (*be*)]] (*env*, *ctx*)
$\equiv$ `Rule` { `id` = *ctr*, `name` = R_n.1 }
`RuleDef`
$\mathcal{E}$[[`not` (*elem.be*) `and` *cond*]]
$\mathcal{A}$[[*sourceColl*$_n$.*reject* := `including`(*sourceColl*$_n$.*reject*, *elem*)]]
$\mathcal{B}$[[*elem*]] (*ctr*, *env*)
$\mathcal{B}$[[*SourceColl*$_n$]] (*ctr*, *env*)
$\mathcal{B}$[[*ctx*]] (*ctr*, *env*)

where *cond* = (*sourceColl*$_n$ `includes` *elem*)
`and` (*sourceColl*$_n$ = *ctx.sourceColl*$_n$)
ctr = *freshId*()
n = *freshRuleNumber*()

Example Translation

```
context University
inv: self.students -> reject(isActive) -> isEmpty()
```

R$_n$.1 `bind` *student* `to` *Student*
`bind` *studentColl*$_n$ `to` *StudentColl*$_n$
`bind` *university* `to` *University*
`ifmatch` *student*, *studentColl*$_n$, *university*
`where` (`not` *isActive*)
`and` (*studentColl*$_n$ `includes` *student*)
`and` (*studentColl*$_n$ = *university.studentColl*$_n$)
`then` *studentColl*$_n$.*reject* := `including` (*studentColl*$_n$.*reject*, *student*)
`end`

R$_n$.2 `bind` *studentColl*$_n$ `to` *StudentColl*$_n$
`bind` *university* `to` *University*
`ifmatch` *studentColl*$_n$, *university*
`where` *studentColl*$_n$_*reject.isEmpty*()
`and` (R_n = `true` `or` R_n = `unknown`)
`then` (R_n := `true`)
`end`

R$_n$.3 `bind` *studentColl*$_n$ `to` *StudentColl*$_n$
`bind` *university* : *University*

```
ifmatch studentColl_n, university
where (not studentColl_n_reject.isEmpty())
then (R_n := false)
end
```

The rule $\texttt{R}_n\texttt{.1}$ in the above example implements the translation scheme for the *reject* operation. This rule results in a intermediate collection value that are used by the other rules. We use a canonical name $studentColl_n\_reject$ to refer to the intermediate collection.

collect

The translation of the *collect* iterator expression results in a collection. This collection is used by another expression to create an invariant.

$\mathcal{E}$[[$se \rightarrow collect$ (be)]] (env, ctx)
$\equiv$ `Rule` { `id` = ctr, `name` = $R_n.1$ }
`RuleDef`
$\mathcal{E}$[[$cond$]]
$\mathcal{A}$[[$sourceColl_n.collect :=$ `including`$(sourceColl_n.collect, elem.be$]]
$\mathcal{B}$[[$elem$)]] (ctr, env)
$\mathcal{B}$[[$SourceColl_n$]] (ctr, env)
$\mathcal{B}$[[ctx]] (ctr, env)

where $cond = (sourceColl_n$ `includes` $elem)$
`and` $(sourceColl_n = ctx.sourceColl_n)$
$ctr = freshId()$
$n = freshRuleNumber()$

Example Translation

The following example shows the use of the *collect* iterator expression within another expression to create an invariant. We can also see that the collection is used as an intermediate value in the translation of the complete expression.

```
context University
inv: self.students -> collect(firstName) -> size() > 0
```

```
R_n.1   bind student : Student
        bind studentColl_n : StudentColl_n
        bind university : University
        ifmatch student, studentColl_n, university
        where ((studentColl_n includes student)
```

```
            and (studentColl_n = university.studentColl_n)
        then studentColl_n.collect := including (studentColl_n.collect, student.firstName)
R_n.2   bind studentColl_n : StudentColl_n
        bind university : University
        ifmatch studentColl_n, university
        where studentColl_n_collect.size() > 0
            and (studentColl_n = university.studentColl_n)
            and (R_n = true or R_n = unknown)
        then (R_n := true)
R_n.3   bind studentColl_n : StudentColl_n
        bind university : University
        ifmatch studentColl_n, university
        where (not studentColl_n_collect.size() > 0)
            and (studentColl_n = university.studentColl_n)
        then (R_n := false)
```

5.3.8 Let Expression

There is no representation of Let expression in RBML. The translation of a Let expresion is equal to the translation of the body expression, where all the occurences of the variables in the body expression are replaced by their definition. A Let declaration is of the form { *var* = *value* }. The operator (⇒) performs the substitution.

The translation scheme for the Let expression is as follows:

$\mathcal{E}$[[let *decls* in *be*]]
≡ $\mathcal{E}$[[*e*]]

wheree = (*decls* ⇒ *be*)

Example Translation

```
context University
inv: let numberOfStudents : Integer = self.students.size()
     in numberOfStudents > 0
```

```
R_n.1   bind university to University
        ifmatch university
        where (self.students.size() > 0)
            and (R_n = true or R_n = unknown)
        then (R_n := true)
        end
```

```
R_n.2   bind university to University
        ifmatch university
        where not (self.students.size() > 0)
        then (R_n := false)
        end
```

The rules `R`$_n$`.1` and `R`$_n$`.2` in the above example implement the translation scheme for the relational expression $>$ (see Section 5.3.2), where the occurence of the variable `numberOfStudents` has been replaced by its definition.

5.3.9 Message Expression

Since there is no representation for the OCL message expression in RBML, we translate the input expression into a *RuleText* element in RBML as described in Section 5.1.2.

The translation scheme is straightforward because we simply translate the input expression into a rule text value.

$$\begin{aligned}&\mathcal{E}[\![\ te \oplus name\ (arg_1, \ldots, arg_n)\ ]\!]\\ &\equiv\ \texttt{Rule}\ \{\ \texttt{id} = ctr,\ \texttt{name} = R_n.1\ \}\\ &\qquad\texttt{RuleText}\ \{\ \texttt{value} = te \oplus name\ (arg_1, \ldots, arg_n)\ \}\end{aligned}$$

where $\oplus\ \in \{\ \text{'}\hat{}\text{'},\ \text{'}\hat{}\hat{}\text{'}\ \}$

5.4 Generating RBML Document

The implementation of the RBML code generator consists of several phases, namely

- processing the code generation environment,
- passing the environment to the nodes of the tree,
- generating the RBML data representation for the OCL constructs, and
- creating an RBML document from the generated data representation.

The following sections explain the process of each phase in detail.

5.4.1 Processing the Environment

In Section 4.7.1, we discussed the process of creating a type environment needed for the type checking process. The type environment contains the subtype relations between types. For the code generation process, we define an environment that is different from the one for the type checking process.

Although these environments are processed from the same model information, we distinguish them in order to simplify the implementation process. As a result, for each process, we provide only the information which are needed.

The environment for the code generation contains

- the classes, their properties (attributes and methods), and their subclasses,
- the associations between classes, and
- the user defined datatypes.

Each element in the environment has a unique *id* associated with it. We use the *id*s from the XMI file which are globally unique, called *uuid*.

Besides the classes from the UML model, we also initialize the environment with two system generated classes, namely:

- The *RBMLBase* class.

  This class owns the rule attributes which are generated in the generic rule based translation approach, as discussed in Section 5.2.2. At the time of the initialization, this class contains no attributes.

- The *Domain* class.

  The primitive types and the collection types in OCL have a number of predefined methods. These methods are not defined in the UML model. Therefore, we introduce a class that acts as a superclass of all the other classes of the model, which is called *Domain* class.

  All the predefined methods in OCL will be the methods of the *Domain* class. When there is a call to the OCL predefined method, we refer to the method of this class.

5.4.2 Passing the Environment

The code generation environment is passed to the nodes of the tree as a chained attribute. We pass it as a chained attribute instead of as an inherited

attribute, because during the process in the sub trees, the environment will be enriched with system generated attributes and classes.

The enriched environment will be passed back to the parent node, where they will be used to generate the RBML document in the remainder of the translation.

5.4.3 Generating the RBML Data Representation

Having all the information needed for the code generation process available, the next step is to generate the RBML data representation for each OCL construct. The data representation is defined in Haskell. The complete data type definition can be seen in Appendix C.

The Haskell data definition for RBML is generated using a package containing a collection of utilities for Haskell and XML, called *HaXml*. The version of HaXml that we use in the implementation is HaXml-1.11 [HaX].

In the paper of Wallace et al. [WR99], two alternatives for representing XML in Haskell are presented. In this implementation, we choose to use a component of HaXml called *DtdToHaskell*, which derives the Haskell data definition based on a specific DTD. Besides the data definition, *DtdToHaskell* also derives appropriate class instance declarations for each generated type. The instance declarations allow us to the read and write a data representation as an XML document.

The first step concerning this implementation is translating the RBML schema into a what we call RBML DTD, and subsequently generate the Haskell data definition from it.

The following AG attribute definition shows a number of attributes needed for generating the code, see AG System User Manual [BSL03] for the complete description about the use of AG.

```
ATTR Expression
     [
      | cid        : Int
        crule      : Int
      | typedexp   : TypedExpression
        premise    : Premise
        action     : Action
        rattr      : String
     ]

ATTR OclFile PackageDeclaration ContextDeclaration
     ContextDeclarations InitOrDerive InitOrDerives
```

```
InvOrDef InvOrDefs PrePostOrBodyDecl PrePostOrBodyDecls
Expression Expressions
[       | gammaG : GammaG | rules: {[Rule]} ]
```

The process for generating the rules is done in the non-terminal `Expression`. In the code generation, we introduce several attributes for passing and collecting the information through the tree.

The chained attribute `cid` is a unique counter for the system generated elements (attribute, condition, rule, binding), while `crule` is a unique counter for the generic rule attribute. The values of `cid` and `crule` are threaded through the tree, and incremented along the way each time we create a new element or a new rule.

Each expression may produce a typed expression, a premise, an action, and/or a rule. The values are passed through the synthesized attributes `typedexp`, `premise`, `action`, and `rattr` to their parent nodes so that they can be used to generate rules.

At the top of the tree `OclFile`, we are only interested in the generated `rules`, and the collected environment `gammaG`.

To get a better idea on how the code generator is implemented, let us look at the code fragment for the *Integer* and *Boolean* literal. The implementation are based on the translation scheme for a literal defined in Section 5.3.1.

```
SEM Expression
   | IntegerLiteral
       lhs.typedexp = TypedExpressionLiteral (str2Literal @value)
       lhs.premise  = noPremise
       lhs.action   = noAction
       lhs.rattr    = voidAttr
       lhs.rules    = []
   | BooleanLiteral
       lhs.typedexp = TypedExpressionLiteral (str2Literal @value)
       lhs.premise  = (premiseC False
                         (unaryCondition (@lhs.cid) False
                                         (unaryLiteral "true"))
                         Nothing )
       lhs.action   = noAction
       lhs.rattr    = voidAttr
       lhs.rules    = []
       lhs.cid      = @lhs.cid + 1
```

The `IntegerLiteral` produces only a literal typed expression, and no premise, action, and rule. The `BooleanLiteral` produces a literal typed expression and a premise, but no action and rule.

The If expression produces several rules in the generation process. The following code implements the translation scheme which is defined in Section 5.3.4.

```
SEM Expression
   | IfExp
       loc.premise = @condition.premise
          .rattr   = 'R':(show @lhs.crule)
       loc.(rules,cid) =
         (  [ rule (@lhs.cid+1) @sid Nothing @premise    -- (R1)
                (actionAssignment
                  (attrAssignment @elements False
                                  @rattr @thenpart.rattr))
            ,rule (@lhs.cid+2) @sid Nothing @negpre     -- (R2)
                (actionAssignment
                  (attrAssignment @elements False
                                  @rattr @elsepart.rattr))
            ] ++
            if isAction @thenpart.action && isAction @elsepart.action
            then
             [  rule (@lhs.cid+3) @sid Nothing @premise  -- (R3)
                     @thenpart.action
               ,rule (@lhs.cid+4) @sid                   -- (R4)
                  (Just (elseOf (@lhs.cid+5)))
                  (premiseC False
                     (unaryCondition (@lhs.cid+6) False
                                     (unaryLiteral "true"))
                     Nothing)
                  @elsepart.action
             ]
            else []
           , if isAction @thenpart.action && isAction @elsepart.action
             then @lhs.cid+7 else @lhs.cid+3 )
       lhs.rules    = @condition.rules ++ @thenpart.rules ++
                      @elsepart.rules ++ @rules
       lhs.cid      = @cid
       lhs.premise  = @premise
       lhs.rattr    = @rattr
       lhs.crule    = @lhs.crule + 1
       lhs.gammaG   = addRule2Env (@rattr,(show @lhs.cid,"RBMLBase"))
```

```
                    (attribute (show @lhs.cid) @rattr "boolean")
                     @lhs.gammaG
```

Due to space limitation, the above code is slightly modified from the actual code. However, they still preserve the main ideas, namely the rule generation.

The If expression has three sub expressions, namely `condition`, `thenpart`, and `elsepart`. We introduce the local attribute `premise` because the value is used several times while generating the rules. The local attribute `rattr` stands for rule attribute, which is also the name of the synthesized attribute of the If node.

To generate a rule, we call a function `rule` and pass the arguments: element id, scope id, optional else-of, premise, and action. The generated rules R1 and R2 correspond to the generic translation approach (the first two rules in the translation scheme). The rules R3 and R4 correspond to the conditional translation approach, where certain conditions have to be met before these rules are generated.

The generated rules, together with the rules from the sub expressions are passed to the parent node. Finally, the new rule attribute is added to the environment `gammaG`.

5.4.4 Creating an RBML Document

The final phases of our code generation process is to create an RBML document from the input OCL expressions. First, we compose a *Rulebase* value from the generated RBML data representation and the collected environments. Then, we write the *Rulebase* value as an XML document conforming to the RBML DTD, which is derived from the RBML schema. The output is statically however to be correct with respect to the DTD [WR99].

Chapter 6

Conclusion

This research project is aimed to design and implement tools that can be used to execute the POWER process. We focus on verifying the conceptual models created within the POWER project. Specifically, we design and implement a tool that type checks UML/OCL models, and then generates RBML document from the well-typed models. The design and implementation of this tool consists of three phases, namely parsing, type checking, and code generation.

A number of shorthands in writing OCL expressions influence the design decisions of our compiler. Although we manage to parse all the OCL constructs, we have set some restrictions in the type checking process (see Section 4.7.5).

We have formalized a set of type rules for OCL. The application of the type rules are shown in some examples of derivation trees. The type rules are also used as our guidance in the implementation of the type checker.

In the code generation to RBML, we introduced two main approaches: the *generic rule based* and the *conditional rule based* translations. We also discussed how each translation should be done, and what the advantages of each are. There are no restrictions on which approach to be chosen since it depends on the purpose of the rule based system. However, we have implemented the two approaches in the code generation process.

The difference in paradigm between OCL, a constraint based language, and RBML, a a rule based language, makes the mapping for some language constructs rather difficult. However, the workarounds taken during the design have been considered carefully so that they preserve the meaning of the initial OCL expressions.

The output of the code generation process is an RBML document. The collection of rules within the RBML document are then validated and verified by the VALENS verification tool. The verification tool checks the consistencies and completeness of the generated rules.

6.1 Contributions

The main contributions of this thesis project are two folds.

Firstly, we have formalized the OCL type system in the form of a set of type rules. The existence of subtype relations in OCL has been incorporated completely while formalizing the type rules. Our implementation of the type checker are able to detect the type errors within OCL expressions, given the UML model information to which these expressions belong.

Secondly, we have successfully implemented the transformation of OCL as a constraint based language into a rule based language RBML. The transformation has been done for a big set of OCL expressions. Some OCL constructs such as message expression, collection literal, and tuple literal are not entirely supported in RBML. The translation of those constructs to RBML are done by generating a free text rule, and free text literals respectively.

6.2 Future Work

The motivation of this project is the generation of knowledge based components from well-typed UML/OCL models. In this project, we present the type checking OCL expressions and the code generation to the rule based system RBML. Although we have implemented the code generator, there are still a lot of things that should be improved.

In the following section, we list some possibilities for future research.

6.2.1 Extending RBML

RBML has limitations in the number of supported constructs. In following section, we list some possible extensions, which makes the translation from OCL to RBML possible or more natural.

Collection Literal

In OCL, one can write a collection literal, such as

```
Set{'red', 'green', 'blue'}
Sequence{2, 4, 6, 8, 10}
```

The collection literals have the notion of lists in many other languages. We can apply many useful operations to the collection literals, such as checking for emptiness, counting the number of elements, joining two collections, etc.

The notion of collection in RBML is introduced by the association with multiplicity greater than 1. However, the collection literals are not supported in RBML. Since the collection literals are very useful, it would be nice to have this feature also in RBML.

Tuple Literal

The OCL tuple literal enables one to group several values into an aggregate value. Each value within the aggregate value has a name attached to it. Consider the following example,

```
context University
def: statistics:Set(TupleType(dept:Department,
                              numTeachers:Integer))
     = departments-> collect( d |
               Tuple{dept:Department = d,
                     numTeachers:Integer = d.member->size()})
```

In the context `University`, we define the attribute `statistics` which record the number of the teachers on each department of a university. The tuple literal is used to group the name of a department and the number of teachers on the department into a single tuple value.

In most languages, the OCL tuple literal is known as a record. There are many situation where the use of OCL tuple literals is helpful. Therefore, having this feature in RBML will be advantageous.

Collecting and Checking Constraints

Referring to our example in Table 5.1 for the unsupported constructs. For example, consider the expression

```
x implies y>0
```

Instead of not supporting the translation for this construct, we could record it as a constraint in a collection of constraints using a certain method. Suppose we have the method `remember` which does the process, the translation will then become

```
Premise: x
Action: remember(y>0)

Premise: not (y>0)
Action: x:=false
```

The method `remember` records the constraint `y>0` in the collection of constraints for the premise `x`. This constraint states that if the value of `x` is *true* then `y` should be greater than 0.

In addition to the method `remember`, we also need a method which checks a constraint against the existing constraints. For example, consider the expression

```
x implies y=-1
```

For the same premise `x`, there is a contradiction with the constraint from the first example about `y`. The generated rules are as follows:

```
Premise: x
Action: checkfor(y=-1)

Premise: not (y=-1)
Action: x:=false
```

We use the method `checkfor` to check the constraint `y=-1`. If the constraint contradicts the existing constraints, the inconsistency is reported.

The bottom line is that predicates are not always *true* or *false*: their truthfulness is simply not yet known. The methods *remember* and *checkfor* allow us to deal with postponing certain checks in a elegant way.

The Action of a Rule Definition

A rule definition in RBML consists of a *premise* and an *action*. The action part supports a limited number of constructs, see Section 5.1. This limitation causes many translations using a *conditional based* approach not to be supported either. We propose that the action is extended to include a method call.

Using a certain method call, we can wrap up any constraint so that it can be used later on to derive other information. For example, the call to the method *remember* discussed in Section 6.2.1. The unsupported construct is passed an argument to the method where it will be recorded in the collection of constraints.

6.2.2 Schema based XML Data Bindings

The existing XML data bindings for Haskell, including HaXml, translate DTDs to Haskell datatypes. The translation of considerably complicated

XML schema is not supported yet. In the technical report by Atanassow et al. [ACJ03], a tool for translating of XML Schema types into Haskell is presented. This tool is implemented as a generic program in Generic Haskell.

Since RBML itself is an XML schema, it is important to study how this tool can be used to translate an XML schema to Haskell datatypes. And how the Haskell datatypes can be read back into an XML document that conforms to the schema.

6.2.3 From RBML to an Executable Code

RBML is not a programming language, instead it is an XML schema to represent rules in the object oriented context. An XML document conforming to RBML schema is referred to as an RBML document. In this project, we have successfully implemented the translation from UML/OCL to RBML. Despite the needs for the extensions discussed in Section 6.2.1, RBML is potentially suitable as a rule based representation system for UML/OCL models. RBML has elements that support both UML and OCL representation.

The most suitable way to test the generated knowledge based components in form of the RBML document, is to translate them to an executable code. The correct translation results in an application that performs automated law enforcement tasks. There is no restriction on which target language to be generated, although some languages will be more suited than others. Object oriented languages, such as Java or C++, are likely possibilities.

Appendix A

OCL Concrete Syntax

The following OCL concrete syntax is created based on the concrete syntax described in OCL 2.0 specification [OMG03]. However, we have extended the grammar specification with the precedence rules for operations. For instance, the multiplicative operators have higher precedence than the additive operators, and the logical operators have higher precedence than the relational operators.

Expressions

```
Expression        := LetExp | ImplicationExp

LetExp            := 'let' VarDecl (',' VarDecl)*
                     'in' Expression

VarDecl           := SimpleName (':' TypeSpecifier)? ('=' Expression)?

ImplicationExp    := LogicalExp ('implies' LogicalExp)*

LogicalExp        := (RelationalEqExp LogicalOp)* RelationalEqExp
LogicalOp         := 'and' | 'or' | 'xor'

RelationalEqExp   := RelationalExp (RelationalEqOp RelationalExp)?
RelationalEqOp    := '=' | '<>'

RelationalExp     := AdditiveExp (RelationalOp AdditiveExp)?
RelationalOp      := '<' | '>' | '>= '| '<='

AdditiveExp       := MultiplicativeExp (AdditiveOp MultiplicativeExp)*
AdditiveOp        := '+' | '-'

MultiplicativeExp := UnaryExp  (MultiplicativeOp UnaryExp)*
MultiplicativeOp  := '*' | '/'

UnaryExp          := UnaryOp PropertyCallExp
                   | PropertyCallExp
```

```
UnaryOp           :=  '-' | 'not'

PropertyCallExp   := AttributeCall
                   | OperationCall
                   | NavigationCall
                   | PathCall
                   | IterateExp
                   | IteratorExp
                   | MessageExp

AttributeCall     := PrimaryExp '.' Name ('@pre')?

OperationCall     := PrimaryExp '.' Name ('@pre')? '(' Expressions? ')'
                   | PrimaryExp '->' Name '(' Expressions? ')'

NavigationCall    := PrimaryExp '.' Name ('@pre')? '[' Expressions? ']'('@pre')?

PathCall          := PrimaryExp ('.' Name '::' Name)*

IterateExp  := PrimaryExp '->' Name '(' (VarDecl ';')? VarDecl '|' Expression ')'

IteratorExp := PrimaryExp '->' Name '(' (VarDecl (','VarDecl)? '|')? Expression ')'

MessageExp        := PrimaryExp ('^^' | '^') Name '('MessageArgs?')'
MessageArgs       := MessageArg (',' MessageArg)*
MessageArg        := '?' (':' Type)?  | Expression

Expressions       := Expression (',' Expression)*

PrimaryExp        := LiteralExp | PropertyCallExp | ParensExp | IfExp

LiteralExp        := StringLiteral
                   | RealLiteral
                   | IntegerLiteral
                   | BooleanLiteral
                   | EnumLiteral
                   | CollectionLiteral

StringLiteral     := '''<String>'''

RealLiteral       := <String>

IntegerLiteral    := <String>

BooleanLiteral    := 'true' | 'false'

EnumLiteral       := PathName '::' SimpleName

TypeSpecifier     := 'Tuple' '(' TypeDeclarations? ')'
                   | CollectionId '(' TypeSpecifier ')'
                   | PathName

TypeDeclarations  := TypeDeclaration (',' TypeDeclaration)*
TypeDeclaration   := SimpleName ':' TypeSpecifier
```

```
CollectionLiteral := CollectionId  '{' CollectionItems? '}'
CollectionId      := 'Collection'| 'Set' | 'OrderedSet' | 'Bag'| 'Sequence'
CollectionItems   := CollectionItem ( ', ' CollectionItem)*
Collection Item   := Expression | Expression '...' Expression

ParensExp         := '(' Expression ')'

IfExp             := 'if' Expression 'then' Expression
                     'else' Expression 'endif'

PathName          := SimpleName ('::' SimpleName)*
SimpleName        := <String>
```

Package and Context Declarations

```
OclFile            := PackageDeclaration

PackageDeclaration := 'package' PathName ContextDeclaration* 'endpackage'
                    | ContextDeclaration+

ContextDeclaration := ClassifierContext | OperationContext | AttrOrAssocContext

ClassifierContext  := 'context' PathName InvOrDef+

OperationContext   := 'context' Operation PrePostOrBodyDecl+

AttrOrAssocContext := 'context' PathName ':' TypeSpecifier InitOrDerive+

InvOrDef           := 'inv' SimpleName? ':' Expression
                    | 'def' SimpleName? ':' DefExpression

DefExpression      := VarDecl '=' Expression
                    | Operation '=' Expression

Operation          :=
       PathName '::' SimpleName '(' FormalParameters? ')' (':' TypeSpecifier)?
     | SimpleName '(' FormalParameters? ')' (':' TypeSpecifier)?

FormalParameters   := FormalParameter (',' FormalParameter)*
FormalParameter    := SimpleName ':' TypeSpecifier

PrePostOrBodyDecl  := ( 'pre' | 'post' | 'body' ) SimpleName? ':' Expression

InitOrDerive       := ( 'init' | 'derive' ) ':' Expression
```

Appendix B

OCL Parsers

```
module OclParser where

import OclSyntax
import OclScanner
import OclTypes
import UU.Parsing
import UU.Scanner.Token
import UU.Scanner.TokenParser
import UU.Scanner.Position

pOclFile :: Parser Token OclFile
pOclFile = OclFile <$> pPackageDeclaration

pPackageDeclaration :: Parser Token PackageDeclaration
pPackageDeclaration =
       PackageDeclaration  <$  pKeyPos "package"
                           <*> pList1Sep (pKey "::") pName
                           <*> pList pContextDeclaration
                           <*  pKey "endpackage"
   <|> ContextDeclarations <$> pList1 pContextDeclaration

pContextDeclaration :: Parser Token ContextDeclaration
pContextDeclaration =
        attrorassoc  <$>  pKeyPos "context"
                     <*> (pNameU <**> pPath )
                     <*   pKey ":"
                     <*>  pTypeSpecifier
                     <*>  pList1 pInitOrDer
    <|> classcontext <$> pKeyPos "context"
                     <*> (pNameU <**> pPath )
                     <*> pList1 pInvOrDef
    <|> opercontext <$> pKeyPos "context"
                    <*> (pNameU <**> pPath )
                    <*> pParens (pListSep (pKey ",") pFormalParameter)
                    <*> pMaybeReturnType
                    <*> pList1 pPrePostOrBodyDecl
   where
```

```
     attrorassoc pos nms tp exprs
       = AttrOrAssocContext pos (init nms) (last nms) tp exprs
     classcontext pos nms decls
       = ClassifierContext pos nms decls
     opercontext pos nms params tp decls
       = OperationContext pos (init nms) (last nms) params tp decls

pPath :: Parser Token (Name -> Names)
pPath =
 (pathf <$ pKey "::" <*> pList1Sep (pKey "::") pNameLU)
   `opt` namef
      where pathf nms nm = (nm:nms)
            namef nm = [nm]

pInitOrDer :: Parser Token InitOrDerive
pInitOrDer = uncurry InitOrDerive <$> pStereoInit
                                  <*  pKey ":"
                                  <*> pExpression

pStereoInit :: Parser Token (Pos, String)
pStereoInit = (\p -> (p, "init")) <$> pKeyPos "init"
          <|> (\p -> (p, "derive")) <$> pKeyPos "derive"

pInvOrDef :: Parser Token InvOrDef
pInvOrDef =
          Invariant <$> pKeyPos "inv"
                    <*> pMaybeName
                    <*  pKey ":"
                    <*> pExpression
      <|> VariableDef <$> pKeyPos "def"
                      <*> pMaybeName
                      <*  pKey ":"
                      <*> pVariableDeclarationDef
                      <*> pExpression
      <|> operdef     <$> pKeyPos "def"
                      <*> pMaybeName
                      <*  pKey ":"
                      <*> (pNameLU <**> pPath)
                      <*> pParens (pListSep (pKey ",") pFormalParameter)
                      <*> pMaybeReturnType
                      <*  pKey "="
                      <*> pExpression
  where
    operdef pos mb nms params tp e
       = OperationDef pos mb (init nms) (last nms) params tp e

pPrePostOrBodyDecl :: Parser Token PrePostOrBodyDecl
pPrePostOrBodyDecl =
   uncurry PrePostOrBodyDecl <$> pStereotype
                             <*> pMaybeName
                             <*  pKey ":"
                             <*> pExpression

pStereotype :: Parser Token (Pos, String)
```

```
pStereotype = (\p -> (p, "pre")) <$> pKeyPos "pre"
          <|> (\p -> (p, "post")) <$> pKeyPos "post"
  <|> (\p -> (p, "body")) <$> pKeyPos "body"

pFormalParameter :: Parser Token FormalParameter
pFormalParameter = FormalParameter <$> pName
                                   <*  pKey ":"
                                   <*> pTypeSpecifier

pInitOrDerive :: Parser Token (Pos, String)
pInitOrDerive = (\p -> (p, "init")) <$> pKeyPos "init"
            <|> (\p -> (p, "derive")) <$> pKeyPos "derive"

pExpression :: Parser Token Expression
pExpression = pLetExp <|> pLogicalImplication

pLetExp :: Parser Token Expression
pLetExp = LetExp <$> pKeyPos "let"
                 <*> pList1Sep (pKey ",") pVariableDeclaration
                 <*  pKey "in"
                 <*> pExpression

pVariableDeclaration :: Parser Token VariableDeclaration
pVariableDeclaration =
   VariableDeclaration <$> pName
                       <*> pMaybeReturnType
                       <*> pMaybeEqExpression

pVariableDeclarationDef :: Parser Token VariableDeclaration
pVariableDeclarationDef =
      (\nm -> VariableDeclaration nm NothingTypeSpecifier
                                     NothingExpression
      ) <$> pName
        <*  pKey "="
   <|>(\nm tp->VariableDeclaration nm (JustTypeSpecifier tp)
                                       NothingExpression
      ) <$> pName
        <*  pKey ":"
        <*> pTypeSpecifier
        <*  pKey "="
   <|>(\nm tp e->VariableDeclaration nm (JustTypeSpecifier tp)
                                        (JustExpression e)
      ) <$> pName
        <*  pKey ":"
        <*> pTypeSpecifier
        <*  pKey "="
        <*> pExpression
        <*  pKey "="

pLogicalImplication :: Parser Token Expression
pLogicalImplication =
   pChainr ((\p e -> BinaryExp p e "implies")<$>(pKeyPos "implies"))
           pLogicalExp
```

```
pLogicalExp :: Parser Token Expression
pLogicalExp =
   pChainl (((\(p,op) re -> BinaryExp p re op) <$> pLogicalOp )
           pRelationalEqExp

pLogicalOp :: Parser Token (Pos, String)
pLogicalOp = (\p -> (p, "and"))<$> pKeyPos "and"
          <|>(\p -> (p, "or")) <$> pKeyPos "or"
  <|>(\p -> (p, "xor"))<$> pKeyPos "xor"

pRelationalEqExp :: Parser Token Expression
pRelationalEqExp = pRelationalExp <**> opt pRelationalEqTailExp id

pRelationalEqTailExp :: Parser Token  (Expression -> Expression )
pRelationalEqTailExp =
   (\(p,op) re -> \le ->BinaryExp p le op re )<$> pRelationalEqOp
                                              <*> pRelationalExp

pRelationalEqOp :: Parser Token (Pos, String)
pRelationalEqOp = (\p -> (p, "="))  <$> pKeyPos "="
             <|> (\p -> (p, "<>")) <$> pKeyPos "<>"

pRelationalExp :: Parser Token Expression
pRelationalExp = pAdditiveExp <**> opt pRelationalTailExp id

pRelationalTailExp :: Parser Token  (Expression -> Expression)
pRelationalTailExp =
   (\(p,op) re -> \le -> BinaryExp p le op re )<$> pRelationalOp
                                               <*> pAdditiveExp

pRelationalOp :: Parser Token (Pos, String)
pRelationalOp = (\p -> (p, ">"))  <$> pKeyPos ">"
            <|>(\p -> (p, "<"))  <$> pKeyPos "<"
     <|>(\p -> (p, ">=")) <$> pKeyPos  ">="
     <|>(\p -> (p, "<=")) <$> pKeyPos "<="

pAdditiveExp :: Parser Token Expression
pAdditiveExp =
   pChainl (((\(p,op) e -> BinaryExp p e op) <$> pAdditiveOp)
           pMultiplicativeExp

pAdditiveOp :: Parser Token (Pos, String)
pAdditiveOp = (\p -> (p, "+")) <$> pKeyPos "+"
           <|>(\p -> (p, "-")) <$> pKeyPos "-"

pMultiplicativeExp :: Parser Token Expression
pMultiplicativeExp =
   pChainl (((\(p,op) e -> BinaryExp p e op) <$> pMultiplicativeOp)
           pUnaryExp

pMultiplicativeOp :: Parser Token (Pos, String)
pMultiplicativeOp = (\p -> (p, "*")) <$> pKeyPos "*"
                 <|>(\p -> (p, "/")) <$> pKeyPos "/"
```

```
pUnaryExp :: Parser Token Expression
pUnaryExp = (\(p,op) e -> UnaryExp p op e ) <$> pUnaryOp
                                            <*> pPropertyCallExp
         <|> pPropertyCallExp

pUnaryOp :: Parser Token (Pos,String)
pUnaryOp = (\p ->(p, "-")) <$> pKeyPos "-"
       <|> (\p ->(p,"not"))<$> pKeyPos "not"

pPropertyCallExp :: Parser Token Expression
pPropertyCallExp = pPrimaryExp
             <??> ( propcall <$> pList1 pPropertyCallExpTail)

propcall fs = \pe -> foldl g pe fs
    where g :: Expression -> (Expression -> Expression) -> Expression
          g exp f2 = f2 exp

pPropertyCallExpTail =
        attrcall <$> pKey "."
                 <*> pName
                 <*> pPre
    <|> opercall <$> pKey "."
                 <*> pName
                 <*> pPre
                 <*> (pParens (pListSep (pKey ",") pExpression))
    <|> navcall  <$> pKey "."
                 <*> pName
                 <*> (pBracks (pListSep (pKey ",") pExpression))
                 <*> pPre
    <|> pathcall  <$> pKey "."
                  <*> pNameU
                  <*  pKey "::"
                  <*> pName
    <|> opercall2 <$> pKey "->"
                  <*> pName
                  <*> (pParens (pListSep (pKey ",") pExpression))
    <|> iterexp   <$> pKey "->"
                  <*> pIterateName
                  <*  pOParen
                  <*> pVarDeclIterate
                  <*> pExpression
                  <*  pCParen
    <|> iterexp  <$> pKey "->"
                  <*> pIteratorName
                  <*  pOParen
                  <*> pVarDeclIterator
                  <*> pExpression
                  <*  pCParen
    <|> msgexp    <$> pMessageOp
                  <*> pName
                  <*> pParens (pListSep (pKey ",")  pMessageArgument)
  where
    attrcall op nm pre = \pe ->AttributeCall pe op nm pre
    opercall op nm pre args = \pe ->OperationCall pe op nm pre args
```

```
    navcall op nm args pre = \pe ->NavigationCall pe op nm args pre
    opercall2 op nm args = \pe ->OperationCall pe op nm False args
    iterexp op nm decls e = \pe ->LoopPropertyCall pe op nm decls e
    msgexp (pos,op) nm args = \pe ->MessageExp pos pe op nm args
    pathcall op nm nm2 =
      \pe ->PathCall pe op (init(nm:[nm2])) (last(nm:[nm2]))

pVarDeclIterate :: Parser Token VariableDeclarations
pVarDeclIterate =
      (\a b -> a:[b])<$> pVariableDeclaration
                         <* (pKey ";" )
                         <*> pVariableDeclaration <* pKey "|"
   <|> (\a -> [a]) <$> pVariableDeclaration <* pKey "|"

pVarDeclIterator :: Parser Token VariableDeclarations
pVarDeclIterator =
     (\a b -> a:[b])<$> pVariableDeclaration
                        <* (pKey "," )
                        <*> pVariableDeclaration <* pKey "|"
 <|> (\a -> [a])       <$> pVariableDeclaration <* pKey "|"
 <|> pSucceed []

pMessageOp  :: Parser Token (Pos, String)
pMessageOp =  (\p -> (p, "^^")) <$> pKeyPos "^^"
          <|> (\p -> (p, "^"))  <$> pKeyPos "^"

pMessageArgument ::Parser Token MessageArgument
pMessageArgument =
      TypeArgument <$> pKeyPos "?" <*> pSucceed "?"
                                   <*> pMaybeReturnType
   <|> ExpArgument <$> pExpression

pPre :: Parser Token Bool
pPre = ((\a b-> True)<$>pKey "@" <*> pKey "pre" ) `opt` False

pPrimaryExp :: Parser Token Expression
pPrimaryExp = pLiteral
          <|> pFeatureCall
          <|> pParens pExpression
          <|> pIfExp
          <|> pVariableExp

pVariableExp  :: Parser Token Expression
pVariableExp = VariableExp <$> pName

pLiteral :: Parser Token Expression
pLiteral = pStringLiteral  <|> pRealLiteral
       <|> pIntegerLiteral <|> pBooleanLiteral
       <|> pEnumLiteral    <|> pCollectionLiteral
       <|> pTupleLiteral

pStringLiteral :: Parser Token Expression
pStringLiteral =(\(s, pos) -> StringLiteral pos s)<$> pStringPos
```

```
pBooleanLiteral :: Parser Token Expression
pBooleanLiteral =
      (\p -> BooleanLiteral p "True")  <$> pKeyPos "true"
   <|>(\p -> BooleanLiteral p "False") <$> pKeyPos "false"

pIntegerLiteral :: Parser Token Expression
pIntegerLiteral = (\(v,p)-> IntegerLiteral p v )<$> pIntegerPos

pRealLiteral :: Parser Token Expression
pRealLiteral = (\(v,p) k i -> RealLiteral p (v++k++i) )
                <$> pIntegerPos
                <*> pKey "."
                <*> pInteger

pEnumLiteral :: Parser Token Expression
pEnumLiteral =
   pNameLU <**> (enumf <$ pKey "::" <*> pList1Sep (pKey "::") pNameLU)
     where enumf nms = \nm->EnumLiteral (init (nm:nms)) (last (nm:nms))

pCollectionLiteral :: Parser Token Expression
pCollectionLiteral =
   uncurry CollectionLiteral
       <$> pCollectionIdentifier
       <*> pCurly (pListSep (pKey ",") pCollectionItem)

pCollectionItem :: Parser Token CollectionItem
pCollectionItem = CollectionRange <$> pExpression
                                  <*  pKey ".."
                                  <*> pExpression
             <|> CollectionExp    <$> pExpression

pCollectionIdentifier:: Parser Token (Pos,String)
pCollectionIdentifier =
       (\p ->(p, "Sequence"))   <$> pKeyPos "Sequence"
   <|> (\p ->(p, "Set"))        <$> pKeyPos "Set"
   <|> (\p ->(p, "Bag"))        <$> pKeyPos "Bag"
   <|> (\p ->(p, "OrderedSet")) <$> pKeyPos "OrderedSet"

pTupleLiteral :: Parser Token Expression
pTupleLiteral =
    TupleLiteral <$> pKeyPos "Tuple"
                 <*> pCurly (pList1Sep (pKey ",") pVariableDeclaration)

pIfExp :: Parser Token Expression
pIfExp = IfExp <$> pKeyPos "if"
               <*> pExpression
               <*  pKey "then"
           <*> pExpression
          <*  pKey "else"
          <*> pExpression
          <*  pKey "endif"

pFeatureCall = pNameLU <**> pFeatureCallAlts
pFeatureCallAlts =
```

```
      propcall3<$> pPre
               <*> (opercall3
                      <$> pParens (pListSep (pKey ",") pExpression)
                        `opt` attrcall3
                    )
  <|> navcall3 <$> pBracks (pListSep (pKey ",") pExpression) <*> pPre
  <|> pathopercall <$ pKey "::"
                   <*> pName
                   <*> pParens (pListSep (pKey ",") pExpression)
  where
    propcall3 pre f = \nm -> f nm pre
    opercall3 args = \nm pre -> ImplicitOperationCall nm pre args
    attrcall3 = \nm pre -> ImplicitAttributeCall nm pre
    navcall3  args pre = \nm -> ImplicitNavigationCall nm args pre
    pathopercall nm2 args =
      \nm ->PathOperationCall (init (nm:[nm2])) (last (nm:[nm2])) args

pName :: Parser Token Name
pName =  (\(nm,pos) -> Name pos nm )<$> pVaridPos

pNameU :: Parser Token Name
pNameU =  (\(nm,pos) -> Name pos nm ) <$> pConidPos

pNameLU :: Parser Token Name
pNameLU = pName <|> pNameU

pIterateName:: Parser Token Name
pIterateName = (\p -> Name p "iterate") <$> pKeyPos "iterate"

pIteratorName:: Parser Token Name
pIteratorName = (\p -> Name p "forAll")<$> pKeyPos "forAll"
           <|> (\p -> Name p "exists") <$> pKeyPos "exists"
           <|> (\p -> Name p "select") <$> pKeyPos "select"
           <|> (\p -> Name p "reject") <$> pKeyPos "reject"
           <|> (\p -> Name p "collect")<$> pKeyPos "collect"

pMaybeName =  mkMaybeParser JustName NothingName pName
pMaybeNameU =  mkMaybeParser JustName NothingName pNameU
pMaybeTypeSpecifier =
   mkMaybeParser JustTypeSpecifier NothingTypeSpecifier pTypeSpecifier
pMaybeExpression =
   mkMaybeParser JustExpression NothingExpression pExpression
pMaybeReturnType =
   mkMaybeParserSep JustTypeSpecifier NothingTypeSpecifier
                    (pKey ":") pTypeSpecifier
pMaybeEqExpression = mkMaybeParserSep JustExpression NothingExpression
                    (pKey "=") pExpression

pTypeSpecifier :: Parser Token TypeSpecifier
pTypeSpecifier =
      TupleType <$> pKeyPos "Tuple"
                <*> pParens (pListSep (pKey ",") pTypeDeclaration)
  <|> (\(pos,nm) tp -> CollectionType (Name pos nm) tp)
                <$> pCollectionIdentifier
```

```
               <*> pParens pTypeSpecifier
  <|> pNameU
      <**> ((pathtp <$ pKey "::" <*> pList1Sep (pKey"::") pNameU)
              `opt` objTp
            )
  where
    pathtp nms = \nm -> PathType (init (nm:nms)) (last (nm:nms))
    objTp      = \nm -> PathType [] nm

pTypeDeclaration :: Parser Token TypeDeclaration
pTypeDeclaration = TypeDeclaration <$> pNameU
                                   <*  pKey ":"
                                   <*> pTypeSpecifier

-----------------
-- Utilities
-----------------

mkMaybeParserSep semJust semNothing sep p
  = semJust <$ sep <*> p <|> pSucceed semNothing

mkMaybeParser semJust semNothing p
  = semJust <$>  p <|> pSucceed semNothing
```

Appendix C

RBML Data Representation

To intermediate the translation from OCL to RBML, we define the following Haskell data representation for RBML:

```
data Action = ActionAttributeValueAssignment AttributeValueAssignment
    | ActionAssociationEstablishment AssociationEstablishment
    | ActionUntypedMethod UntypedMethod
    | ActionArithmeticExpression ArithmeticExpression

data Association = Association Association_Attrs Source Destination
                               (Maybe Role) (Maybe InverseRole)

data Association_Attrs = Association_Attrs
    { associationId :: String
    , associationName :: String
    , associationComment :: String
    }
data AssociationEstablishment =
   AssociationEstablishment AssociationEstablishment_Attrs
                            RoleReference InverseRoleReference

data AssociationEstablishment_Attrs = AssociationEstablishment_Attrs
    { associationEstablishmentObjectqueryexpression :: String }

data Argument = Argument Argument_Attrs (Maybe Type)

data Argument_Attrs = Argument_Attrs
    { argumentId :: String
    , argumentName :: String
    , argumentComment :: String
    , argumentArgumenttype :: String
    , argumentDefaultvalue :: String
    }

data ArgumentRef = ArgumentRef { argumentRefIdref :: String }
```

```
newtype ArgumentExpression = ArgumentExpression TypedExpression

data ArithmeticExpression =
     ArithmeticExpression LeftHand ArithmeticOperator RightHand

data ArithmeticOperator = ArithmeticOperator
    { arithmeticOperatorOperator :: ArithmeticOperator_operator }

data ArithmeticOperator_operator =
     ArithmeticOperator_operator_plus
   | ArithmeticOperator_operator_minus
   | ArithmeticOperator_operator_times
   | ArithmeticOperator_operator_divided_by
   | ArithmeticOperator_operator_int_div
   | ArithmeticOperator_operator_mod

data Attribute = Attribute Attribute_Attrs (Maybe Type)

data Attribute_Attrs = Attribute_Attrs
    { attributeId :: String
    , attributeName :: String
    , attributeComment :: String
    , attributeAccess :: String
    , attributeStatic :: String
    , attributeInitialvalue :: String
    , attributeConstant :: String
    }

data AttributeReference = AttributeReference (Maybe Qualifier) AttributeRef

data AttributeRef = AttributeRef { attributeRefIdref :: String }

data AttributeValue = AttributeValue AttributeValue_Attrs AttributeRef

data AttributeValue_Attrs = AttributeValue_Attrs
    { attributeValueId :: String
    , attributeValueName :: String
    , attributeValueComment :: String
    , attributeValueValue :: String
    }

data AttributeValueAssignment =
     AttributeValueAssignment AttributeValueAssignment_Attrs
                              AttributeReference TypedExpression

data AttributeValueAssignment_Attrs =
  AttributeValueAssignment_Attrs {attributeValueAssignmentRetractable::String}

data BinaryLogicalExpression =
   BinaryLogicalExpression BinaryLogicalExpression_Attrs LeftHand RightHand

data BinaryLogicalExpression_Attrs = BinaryLogicalExpression_Attrs
  { binaryLogicalExpressionOperator :: BinaryLogicalExpression_operator }
```

```
data BinaryLogicalExpression_operator =
     BinaryLogicalExpression_operator_eq
   | BinaryLogicalExpression_operator_gt
   | BinaryLogicalExpression_operator_ge
   | BinaryLogicalExpression_operator_lt
   | BinaryLogicalExpression_operator_le
   | BinaryLogicalExpression_operator_ne
   | BinaryLogicalExpression_operator_includes
   | BinaryLogicalExpression_operator_elementof

data BinaryLogicalOp = BinaryLogicalOp BinaryLogicalOp_Attrs Condition

data BinaryLogicalOp_Attrs = BinaryLogicalOp_Attrs
    { binaryLogicalOpValue :: BinaryLogicalOp_value }

data BinaryLogicalOp_value = BinaryLogicalOp_value_and
                           | BinaryLogicalOp_value_or

data BinaryLogicalOperator =
     BinaryLogicalOperatorCondition BinaryLogicalOperator_Attrs Condition
   | BinaryLogicalOperatorPremise BinaryLogicalOperator_Attrs Premise

data BinaryLogicalOperator_Attrs = BinaryLogicalOperator_Attrs
    { binaryLogicalOperatorValue :: BinaryLogicalOperator_value }

data BinaryLogicalOperator_value = BinaryLogicalOperator_value_and
                                 | BinaryLogicalOperator_value_or

data Binding = Binding Binding_Attrs ClassRef

data Binding_Attrs = Binding_Attrs
    { bindingId :: String
    , bindingName :: String
    }

data BoolLiteral = BoolLiteral
    { boolLiteralFreeformat :: String
    , boolLiteralPredefined :: String
    }

data BooleanAttribute = BooleanAttribute (Maybe Qualifier) AttributeRef

data BooleanMethod = BooleanMethod CalledMethod [PassedArgument]

data BooleanEnumValue = BooleanEnumValue (Maybe Qualifier) EnumValueRef

newtype Calculation = Calculation TypedExpression

data CalledMethod = CalledMethod (Maybe Qualifier) Method

data Class = Class Class_Attrs [Attribute]
                   [(OneOf2 PrimitiveMethod DomainMethod)] [Instance] [SubClass]

data Class_Attrs = Class_Attrs
```

```
    { classId :: String
    , className :: String
    , classComment :: String
    }

data ClassRef = ClassRef { classRefIdref :: String }

data Condition = Condition Condition_Attrs
                       (OneOf2 UnaryLogicalExpression BinaryLogicalExpression)
                       (Maybe BinaryLogicalOp)

data Condition_Attrs = Condition_Attrs
    { conditionId :: String
    , conditionNot :: String
    }

data Constraint = Constraint Constraint_Attrs (List1 Range)

data Constraint_Attrs = Constraint_Attrs
    { constraintId :: String
    , constraintName :: String
    , constraintComment :: String
    }

data DataType =
     DataType DataType_Attrs (Maybe Parent) (OneOf2 [Constraint] [EnumValue])

data DataType_Attrs = DataType_Attrs
    { dataTypeId :: String
    , dataTypeName :: String
    , dataTypeComment :: String
    , dataTypePrimitivetype :: DataType_primitivetype
    }

data DataType_primitivetype = DataType_primitivetype_boolean
                            | DataType_primitivetype_string
                            | DataType_primitivetype_integer
                            | DataType_primitivetype_real
                            | DataType_primitivetype_date
                            | DataType_primitivetype_time

data DataTypeRef = DataTypeRef { dataTypeRefIdref :: String }

data Destination = Destination
    { destinationIdref :: String
    , destinationMultiplicity :: String
    }

data DomainMethod =
     DomainMethod DomainMethod_Attrs (Maybe Type) [Argument]
                (OneOf5 GetAccessor SetAccessor Calculation Premise Action)

data DomainMethod_Attrs = DomainMethod_Attrs
    { domainMethodId :: String
```

```
    , domainMethodName :: String
    , domainMethodComment :: String
    , domainMethodAccess :: String
    , domainMethodStatic :: String
    }

data ElseOf = ElseOf { elseOfIdref :: String }

data EnumValue = EnumValue
    { enumValueId :: String
    , enumValueAlias :: String
    , enumValuePredefinedvalue :: String
    , enumValueFreeformatvalue :: String
    , enumValueComment :: String
    }

data EnumValueReference = EnumValueReference (Maybe Qualifier) EnumValueRef

data EnumValueRef = EnumValueRef { enumValueRefIdref :: String }

data GetAccessor = GetAccessor (Maybe Qualifier) AttributeRef

data Goal = Goal AttributeReference RoleReference InverseRoleReference

data Inference =
     Inference Inference_Attrs Scope [PostedRuleSet] [Goal] [PostedRule]

data Inference_Attrs = Inference_Attrs
    { inferenceId :: String
    , inferenceName :: String
    , inferenceChainmode :: String
    , inferenceComment :: String
    }

data Instance = Instance Instance_Attrs [AttributeValue]

data Instance_Attrs = Instance_Attrs
    { instanceId :: String
    , instanceName :: String
    , instanceComment :: String
    }

data InverseRole = InverseRole
    { inverseRoleId :: String
    , inverseRoleName :: String
    , inverseRoleComment :: String
    }

data InverseRoleReference =
     InverseRoleReference (Maybe Qualifier) InverseRoleRef

data InverseRoleRef = InverseRoleRef { inverseRoleRefIdref :: String }

newtype LeftHand = LeftHand TypedExpression
```

```
data Literal = Literal
    { literalFreeformat :: String
    , literalPredefined :: String
    }

data Method = Method { methodIdref :: String }

data Parent = Parent { parentIdref :: String }

data PassedArgument = PassedArgument ArgumentRef ArgumentExpression

data PostedRule = PostedRule
    { postedRuleIdref :: String
    , postedRulePriority :: String
    }

data PostedRuleSet = PostedRuleSet
    { postedRuleSetIdref :: String
    , postedRuleSetPriority :: String
    }

data Premise =  Premise Premise_Attrs (OneOf2 Condition Premise)
                        (Maybe BinaryLogicalOperator)

data Premise_Attrs = Premise_Attrs { premiseNot :: String }

data PrimitiveMethod =
     PrimitiveMethod PrimitiveMethod_Attrs Type [Argument]

data PrimitiveMethod_Attrs = PrimitiveMethod_Attrs
    { primitiveMethodId :: String
    , primitiveMethodName :: String
    , primitiveMethodComment :: String
    , primitiveMethodAccess :: String
    , primitiveMethodStatic :: String
    , primitiveMethodImplementation :: String
    }

data PrimitiveType = PrimitiveType { primitiveTypeName :: String }

data Qualifier = QualifierTypedMethod TypedMethod
               | QualifierAttributeReference AttributeReference
               | QualifierClassRef ClassRef
               | QualifierRoleReference RoleReference
               | QualifierInverseRoleReference InverseRoleReference
               | QualifierDataTypeRef DataTypeRef

data Range = Range
    { rangeStartvalue :: String
    , rangeStartvalueincluded :: String
    , rangeEndvalue :: String
    , rangeEndvalueincluded :: String
    }
```

```
newtype RightHand = RightHand TypedExpression

data Role = Role
    { roleId :: String
    , roleName :: String
    , roleComment :: String
    }

data RoleReference = RoleReference (Maybe Qualifier) RoleRef

data RoleRef = RoleRef { roleRefIdref :: String }

data Rule = Rule Rule_Attrs Scope RuleDef [Binding]

data Rule_Attrs = Rule_Attrs
    { ruleId :: String
    , ruleName :: String
    }

data RuleDef = RuleDef (Maybe ElseOf) Premise Action

data RuleSet = RuleSet RuleSet_Attrs Scope [PostedRule] [PostedRuleSet]

data RuleSet_Attrs = RuleSet_Attrs
    { ruleSetId :: String
    , ruleSetName :: String
    , ruleSetComment :: String
    }

data Rulebase = Rulebase Rulebase_Attrs (List1 Class) [DataType]
                         [Association] (List1 Rule) [RuleSet] [Inference]

data Rulebase_Attrs = Rulebase_Attrs
    { rulebaseVersion :: String
    , rulebaseName :: String
    , rulebaseComment :: String
    }

data Scope = Scope { scopeIdref :: String }

data SetAccessor = SetAccessor (Maybe Qualifier) AttributeRef

data Source = Source
    { sourceIdref :: String
    , sourceMultiplicity :: String
    }

data SubClass = SubClass
    { subClassId :: String
    , subClassName :: String
    , subClassComment :: String
    }
```

```
data Type = TypeDataTypeRef Type_Attrs DataTypeRef
          | TypePrimitiveType Type_Attrs PrimitiveType
          | TypeClassRef Type_Attrs ClassRef

data Type_Attrs = Type_Attrs { typeMultiplicity :: String }

data TypedMethod = TypedMethodCalledMethod CalledMethod
                 | TypedMethodPassedArgument [PassedArgument]

data TypedExpression =
     TypedExpressionArithmeticExpression ArithmeticExpression
   | TypedExpressionAttributeReference AttributeReference
   | TypedExpressionTypedMethod TypedMethod
   | TypedExpressionLiteral Literal
   | TypedExpressionEnumValueReference EnumValueReference
   | TypedExpressionRoleReference RoleReference
   | TypedExpressionInverseRoleReference InverseRoleReference

data UnaryLogicalExpression =
     UnaryLogicalExpressionBoolLiteral BoolLiteral
   | UnaryLogicalExpressionBooleanAttribute BooleanAttribute
   | UnaryLogicalExpressionBooleanMethod BooleanMethod
   | UnaryLogicalExpressionBooleanEnumValue BooleanEnumValue

data UntypedMethod = UntypedMethod CalledMethod [PassedArgument]
```

Bibliography

[ACJ03] Frank Atanassow, Dave Clarke, and Johan Jeuring. *Scripting XML with Generic Haskell*. Technical Report UU-CS-2003-023, Utrecht University, 2003.

[Ass] Computer Associates. *CleverPath Aion Business Rules Expert*. http://supportconnectw.ca.com/public/aion/aionsupp.asp. Manuals.

[BJR00] Grady Booch, Ivar Jacobson, and James Rumbaugh. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison Wesley Longman, Inc., 2nd edition, 2000.

[BSL03] Arthur Baars, Doaitse Swierstra, and Andres Löh. *UU AG System User Manual*. Department of Computer Science, Utrecht University, September 2003.

[Car97] Luca Cardelli. *Type Systems*, chapter 103, pages 2208–2236. CRC Press, 1997. Handbook of Computer Science and Engineering.

[CW85] L. Cardelli and P. Wegner. *On Understanding Types, Data Abstraction, and Polymorphism. Computing Surveys*, 17(4), 1985.

[CW02] Tony Clark and Jos Warmer. *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*. Springer-Verlag Berlin Heidelberg, 2nd edition, 2002. Lecture notes in Computer Science; 2263.

[DHO01] Birgit Demuth, Heinrich Hussmann, and Sven Obermaier. *Experiments with XMI Based Transformations of Software Models*, 2001.

[DS03] Atze Dijkstra and S. Doaitse Swierstra. *Implementation of Programming Languages*. Institute of Information and Computing Science, Utrecht University, February 2003. Lecture Notes.

[EK98] T. Van Engers and P. Kordelaar. *POWER: Programme for an Ontology based Working Environment for modeling and use of Regulations and legislation.* In *Proceedings of the ISMICK'99*, 1998.

[Fin00] Frank Finger. *Design and Implementation of a Modular OCL Compiler.* Master's thesis, Dresden University of Technology, 2000.

[Fok02] D. Moude Foko. *A Compiler Implementation for UML/OCL.* Master's thesis, Utrecht University, 2002.

[HaX] *HaXml.* http://www.cs.york.ac.uk/fp/HaXml/.

[JS01] Johan Jeuring and Doaitse Swierstra. *Grammars and Parsing.* Institute of Information and Computing Science, Utrecht University, 2001. Lecture Notes.

[Liba] LibRT. *VALENS.* http://www.librt.com/products/valens.html.

[Libb] LibRT. *XML Schemas.* http://www.librt.com/products/XML-schemas.html.

[oB] University of Bremen. *A UML-based Specification Environment.* http://dustbin.informatik.uni-bremen.de/projects/USE/.

[OMG03] OMG. *Response to the UML 2.0, OCL RfP (ad/2000-09-03)*, January 2003. Revised Submission, Version 1.6.

[oT] Dresden University of Technology. *Dresden OCL Toolkit.* http://dresden-ocl.sourceforge.net/index.html.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages.* The MIT Press, 1st edition, 2002.

[Ric02] Mark Richters. *A Precise Approach to Validating UML Models and OCL Constraints.* PhD thesis, University of Bremen, 2002.

[Swi] Doaitse Swierstra. *Utrecht Parser Combinators: Fast, Error Repairing Parser Combinators.* Haskell Libraries.

[vGB+01] Tom M. van Engers, Rik Gerrits, Margherita Boekenoogen, Erwin Glassee, and Patries Kordelaar. *POWER: using UML/OCL for modeling legislation.* In *Proceedings of the 8th international conference on Artificial intelligence and law*, pages 157–167. ACM Press, 2001. An application report.

[WK99] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison Wesley Longman, Inc., 1st edition, 1999.

[WR99] Malcolm Wallace and Colin Runciman. *Haskell and XML: Generic Combinators or Typed-Based Translation?* In *Proceedings of the International Conference on Functional Programming, Paris, Sept 1999*, 1999.