CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY

ASSIGNMENT OF BACHELOR'S THESIS

Title:	OpenPonk: an Implementation of a Parser and Interpreter of OCL
Student:	Jakub Svoboda
Supervisor:	Ing. Robert Pergl, Ph.D.
Study Programme:	Informatics
Study Branch:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2018/19

Instructions

1. Acquaint yourself with the OCL language, the Pharo language and environment and the OpenPonk platform.
2. Design and implement an OCL parser for Pharo, including appropriate tests.
3. Integrate the parser into the OpenPonk platform.
4. Demonstrate the parser on a case study of ensuring OntoUML model constraints.

References

http://pharo.org
https://openponk.github.io
https://modeling-languages.com/ocl-tutorial/
http://www.omg.org/spec/OCL/

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague October 31, 2017

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

OpenPonk: an Implementation of a Parser and Interpreter of OCL

Jakub Svoboda

Department of Computer Science

Supervisor: Ing. Robert Pergl, Ph.D.

May 14, 2018

Acknowledgements

I would like to sincerely thank my supervisor, Ing. Robert Pergl, Ph.D., for providing me with valuable feedback and guiding me throughout the whole implementation and writing process. I am also grateful to Bc. Peter Uhnák for helping me with OpenPonk and Pharo. Finally I owe my deepest gratitude to my parents, Ivo and Zuzana, for their never-ending support.

Declaration

In Prague on May 14, 2018 . . . .

Citation of this thesis

Svoboda, Jakub. *OpenPonk: an Implementation of a Parser and Interpreter of OCL*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

Tato bakalářská práce se zabývá návrhem a implementací syntaktické analýzy a interpreta OCL jazyka pro OpenPonk modelovací platformu v prostředí Pharo. Nejdříve se v práci popisuje OCL jazyk, algoritmy syntaktické analýzy, nástroje pro syntaktickou analýzu, které jsou dostupné ve Pharo prostředí (PetitParser, SmaCC). Dále se práce zabývá analýzou problémů a řešení, které vedly k výsledné implementaci OCL interpreta. Nakonec se funkčnost OCL interpreta ukáže na dodržování OntoUML modelových omezení.

Klíčová slova OCL, parser, interpret, Pharo, Openponk, OntoUML, SmaCC, PetitParser

Abstract

This bachelor thesis covers the creation of an OCL parser and interpreter for OpenPonk modeling platform in the Pharo environment. We describe the OCL language, parsing algorithms and parsing frameworks avaiable in Pharo (PetitParser, SmaCC). We then analyze problems and approaches that led to the final implementation of the OCL interpreter. In the end we show the interpreter functionality on OntoUML metamodel constraints.

Keywords OCL, parser, interpreter, Pharo, Openponk, OntoUML, SmaCC, PetitParser

Contents

List of Figures

List of Tables

Introduction

Conceptual modeling is an important part of the analysis, design and implementation life-cycle of any moderately complex software product. Things being modeled range from business processes, use cases, database and class architectures to system overviews and more. One of the most widespread standard for creating these models and diagrams is the Unified Modeling Language (UML) developed by the Object Managment Group (OMG).

One problem with UML (and basically any graphical notation) is that there is no way to express some of the more complex relationships, constraints or invariants that are important in the system or concept being modeled. The answer to this problem is the Object Constraint Language (OCL) also developed by OMG. It enables the modeler to specify conditions and invariants on the model using expressions and context declarations. These can range from simple conditions on instance models to complex definitions and invariants performed on metamodels.

There are many applications that help with the creation of UML models. One of them is the OpenPonk modeling platform that is implemented in Pharo. OpenPonk is being developed by the Center for Conceptual Modeling and Implementation (CCMi) group at the Faculty of Information Technology at the Czech Technical University. What the OpenPonk platform lacks in its current state, is an OCL interpreter that could evaluate OCL expressions and constraints. Which brings us to the goal of this thesis…

Goal

The goal of this thesis is to implement a functional OCL interpreter in the Pharo environment. Then integrate the interpreter in the OpenPonk modeling platform and showcase the interpreter on a case study of ensuring OntoUML model constraint. After a discussion with the thesis supervisor, we came to a conclusion that it is enough to create an interpreter that is capable of parsing and evaluating a subset of all the OCL language expressions. Creating an interpreter, that could parse and evaluate any and all OCL expressions and context declarations, would be too time consuming and out of the scope of this thesis. What is necessary is to show that creating a fully featured interpreter can be reasonably accomplished if given enough time and provide a foundation that could be extended upon.

Thesis Structure

In this section we look on the overall structure of the thesis and a short description of the topics for each chapter.

Object Constraint Language

In this chapter we shortly describe the purpose of OCL. We look into what OCL expressions are possible, what context declarations there are and what is their function. Finally we look into the OCL standard Library which has to be avaiable for any OCL interpreter to be successfully implemented. The purpose of this chapter is for the reader to get familiar with the OCL language good enough, so that he or she is capable of reading and telling the meaning of OCL expressions or context declarations.

Parsing

In this chapter we learn about the basic theory necessary to understand what is going on behind the scenes of parsers and parser frameworks. We learn about the more traditional Left to right, Leftmost derivation (LL) parsers and Left to right, Rightmost derivation (LR) parsers and LR variants like Look-Ahead LR (LALR) parsers and Generalized LR (GLR) parsers. We also look into some alternative methodologies.

Pharo

Here we briefly introduce Pharo and its environment. We mainly go over its syntax and some of the development tools.

Pharo's parser frameworks

In this chapter we get familiar with the two avaiable parser frameworks used in Pharo, PetitParser and Smalltalk Compiler-Compiler (SmaCC). We look into how they are used and what their debugging tools are. We go a little more in depth with SmaCC as it comes with its own syntax for writing a language grammar.

Analysis and design

Here we go over the thought process that lead to the final design and implementation of the OCL interpreter. We talk about approaches that were tried during the implementation and why some of them did not work.

Implementation

In this chapter we look on the final implementation architecture. We go over the packages that the implementation is divided in and the classes these packages contain and how they are used. After that follows an overview of the interpretation process.

Testing

In the final chapter we discuss unit and integration tests that were implemented. We also describe OntoUML and demonstrate the OCL interpreter functionality on upholding an OntoUML metamodel constraint of our selection.

Part I

Review

Chapter 1

Object Constraint Language

To implement an interpreter for the OCL language we first must know what the OCL language is. What does the syntax looks like? What expressive power does the language offer us? To answer these questions we will introduce the OCL language in this chapter. We will go over the different kinds of the language expressions and context declarations. We will also introduce some basic types, collection types and predefined iterators which make up the standard library. In this chapter I have been using information from the OCL specification [16], other sources will be explicitly referenced.

The OCL language is used with UML models as a complement to help specify constraints like context invariants and operation preconditions and postconditions. It can also specify the results and bodies of operations and specify how a certain attribute's value can be derived from the model. With this a designer can more concretely specify aspects of the system that would be impossible with pure UML. This can then help in model-driven engineering (MDE), better quality code generation or expressing well-formdness rules. It helps solve the natural langugage ambiguity and the graphical notation limitations. [7] [8]

The OCL language is a declarative language without side-effects, that means that a model is in the same state as it was before evaluating any OCL code.

1.1 Expressions

Expressions are a way of referencing or computing a value. Every expression has a type. They are used in all of the context declarations.

1.1.1 Literal Expressions

Literals directly represent a value. They are instances of some type, or in the case of a type literal expression they can also be a reference to an instance of a metatype. There are numeric, boolean, null, invalid, string, collection, enum and type literals.

Example: `5, true, 'string', 90.4, null, Integer`

1.1.2 Navigation Calls

Navigation calls are used to access attributes and associations of objects and types. Their value and type is computed from the attribute value of the current instance.

Example: `anObject.attribute, aCollection->attribute`

1.1.3 Operation Calls

Since OCL is a side-effect free language, operation calls cannot call just any operation. The only permitted operations are queries, which are side-effect free. Every operation has a return type, which is the resulting type of the operation call. The value of the operation call is the resulting value of the query. Operations can have one or more arguments, which is a comma separated list of expressions.

Example: `anObject.operation(), aCollection->operation(4, true)`

1.1.4 Collection Iterators

Collection iterators provides us with the ability to perform high-level iterations on collections. Iterator expressions have two optional iterator variables. Some iterator expression require two to be explicitly declared. Most iterator expressions work with one iterator variable, which can be implicit. The iterator expressions are predefined in the standard library. The resulting value and type depends on the return value and type of the iterator itself, also defined in the standard library.

Example: `aCollection->collect(iterator | iterator.attribute)`

1.1.5 Let Expression

The let expression allows us to define helper variables for one following expression. The defined variables only exist in the scope of this following expression. Also the type and value of the let expression is the resulting type and value of the following expression.

Example: `let x: Integer = 0 in x + x`

1.1.6 If Expression

The if expression allows us to do control flow. It evaluates the condition expression which must conform to Boolean (Type conformance is described in 1.4). If the condition expression's value is true, the resulting value and type are computed from the then expression, otherwise they are calculated from the else expression.

Example: `if true then 1 else 0 endif`

1.1.7 Message Expression

A special expression that is avaiable only in operation postconditions (discussed in subsection 1.2.2). There are two variants of the message expressions, one with the following syntax: `object^operate(1, true)`, which results in `true` if the operation `operate` was called with the arguments `1` and `true` on the object during the execution of the context operation, `false` otherwise. Second variant is the `object^^operate(1, true)`, which returns a collection of a special OCL type OclMessage. The OclMessage can be then queried for result information for example.

1.1.8 @pre

The @pre expression can also be only used in operation postconditions. If we want to access an attribute's original value, before the context operation was called, we just write the `@pre` after the attribute name.

Example: `anObject.speed @pre`

1.2 Context Declarations

Context declarations are used to specify some constraints or define some additional attributes or operations on a context. The context can be a domain-specific element or it can be a metamodel-specific element like a Classifier or an Operation. The context declarations are bound to this element which is referred to as the context type.

1.2.1 Invariants

Context invariants are used to define some constraints on a context type. They are used to constraint the type's attributes and and the attributes of any reachable association. For a context invariant to be true the constraint must be valid for every instance of the type. The invariant expression's type must conform to Boolean.

```
context ContextName
inv OptionalInvariantName: OCLExpression
```

Listing 1.1: Context invariant declaration

1.2.2 Operation Conditions

Operation pre- and postconditions are used to define operation contracts. What is specified in a precondition must be true when the operation is being called. The postcondition must be true when the given operation returns. Each operation can have more than one pre- and postcondition defined. The conditions can have an optional name. The condition expression's type must conform to Boolean.

```
context ContextName::OperationName(ArgumentName: Type, ...): Type
pre OptionalPreName: OCLExpresion
post OptionalPostName: OCLExpression
```

Listing 1.2: Operation pre- and postcondition

1.2.3 Operation Bodies

Operation body is used to specify the result of an operation. Since the whole body is an OCL expression, that means that the operation is without side effects and thus is a query. Because queries can be used in OCL expressions, any operation with a defined operation body can be used in other OCL expressions as well. The type of the result must conform to the operation's return type.

```
context ContextName::OperationName(ArgumentName: Type, ...): Type
body: OCLExpression
```

Listing 1.3: Body declaration

1.2.4 Initial and Derived Values

With this context decleration we can specify initial and derived values for attributes of the context type. With an initial value we specify what value the attribute must have upon creation of the context type instance. With a derived value we specify the value of the attribute at any given time. An attribute can have both initial and derived value specified, but in that case they must not be contradictory. The initial and derived expression's type must conform to the attribute's type.

```
context ContextName::AttributeName: Type
init: OCLExpression
derive: OCLExpression
```

Listing 1.4: Initial and derived value declaration

1.2.5 Operation and Attribute Definitions

Definitions allow us to define a helper attribute or an operation on the context type. Definitions can be instance scoped or static.

```
context ContextName
def: AttributeName: Type = OCLExpression
static def: OperationName(ArgumentName: Type, ...): Type = OCLExpression
```

Listing 1.5: Attribute and operation defined on Classifier

1.3 Standard Library

The standard library defines number of types and their operations. For collection types it also defines a number of predefined iterators. Every type is an instance of some metatype. Any complete implementation of the OCL language must include this library.

1.3.1 Basic Types

1.3.1.1 Real

Represents a real number. Instance of the PrimitiveType metatype (from UML).

Example operations: `+, -, *, /, floor()`

1.3.1.2 String

Represents a sequence of characters. The characters can consist of non-Roman alphabet characters. Instance of the PrimitiveType metatype (from UML).

Example operations: `concat(), size(), substring()`

1.3.1.3 Boolean

Represents a boolean. The only two instances are `true` and `false`. Instance of the PrimitiveType metatype (from UML).

Example operations: `and, not, xor, or, implies`

1.3.1.4 Integer

Represents an integer number. Instance of the PrimitiveType metatype (from UML).

Example operations: `+, -, *, /, abs()`

1.3.1.5 UnlimitedNatural

Represents any whole non-negative number. Used to represent the multiplicity of an association. It differs from an Integer in that it can hold the value `*`, which means unlimited. Instance of the UnlimitedNaturalType metatype.

Example operations: `+, /, *`

1.3.1.6 OclAny

OclAny acts as supertype to all the other types in the way that all types conform to it and also any property that is defined on OclAny is also defined on all the other types, without them actually inheriting from OclAny. OclAny then represents any object. Instance of the AnyType metatype.

Example operations: `oclIsKindOf(), oclType(), oclAsType()`

1.3.1.7 OclInvalid

Contradictory to OclAny, the OclInvalid type is a type that conforms to all the other types. The only instance is `invalid`. The rule is that any navigation or operation call on `invalid` results in `invalid`. The operations defined on OclAny and logical operators are an exception to this rule. Represents an invalid value that can be the result of some error, e.g. not all types in an expression conform correctly (e.g. we cannot compare a Boolean with an Integer). Instance of the InvalidType metatype.

1.3.1.8 OclMessage

Represents the result of calling and operation on or sending a signal to an object. The type itself is templated. The types are specified with the operation. Because there is an infinite number of possible operation signatures, there is also an infinite number of the OclMessage types. Instance of the metatype MessageType.

Example operations: `hasReturned(), result(), isOperationCall(), isSignalSent()`

1.3.1.9 OclVoid

OclVoid is a type that conforms to all the other types, except OclInvalid. The only instance is `null`. As with OclInvalid any navigation or operation call performed on `null` results in `invalid`, except the properties defined on OclAny and the logical operators. Represents an unassigned or undefined value.

1.3.2 Collection Types

All the collection types are parametrized template types. They have an element type template argument. The element type can be any other type including collections themselves. That means that all the possible collection types cannot be instantiated at once, since we can keep recursively defining the element type as some collection type.

1.3.2.1 Collection(T)

An abstract superclass to the four other collection types. Defines common attributes and operations. Instance of the metatype CollectionType.

Example operations: `size(), includes(), isEmpty()`

1.3.2.2 Set(T)

Represents a mathematical set, i.e. no element is contained in the set more than once. Instance of the metatype SetType.

1.3.2.3 Bag(T)

A bag can contain more than one occurence of the same element and does not order its elements in any way. Instance of the metatype BagType.

1.3.2.4 Sequence(T)

The Sequence type allows duplicates, but orders its elements. Instance of the metatype SequenceType.

1.3.2.5 OrderedSet(T)

A set whose elements are ordered. Altough it shares the properties of a Set and a Sequence it inherits from neither. Inherits from Collection. Instance of the metatype OrderedSetType.

1.3.3 Predefined Iterators

Important part of the standard library is the definition of iterators for the collection types. As have been previously stated they allow us to do high-level iteration operations on collections and except a few(like the `union`) most are defined on all collection types. Some examples of the predefined iterators: `any, select, reject, collect, exists, forAll, isUnique, union`.

1.4 Type Conformance

OCL is a statically typed language and each expression has its own type. An expression where all the types conform is a valid expression. When a type A conforms to a type B it means that the instance of type A can be substituted in place of an instance of type B.

The type conformance rules for domain specific classes are the following:

- A type conforms to itself
- A type conforms to its supertype
- Type conformance is transitive, i.e. if type T1 conforms to type T2 and T2 conforms to type T3, then T1 conforms to T3

Collection types Set(T1), OrderedSet(T1), Bag(T1) and Sequence(T1) conform to any Collection(T2) under the condition that T1 conforms to T2.

Integer conforms to Real. UnlimitedNatural conforms to Integer, except when the value of the UnlimitedNatural is `*`.

Chapter 2

Parsing

An interpreter is a program that reads a source code of some program and executes operations depending on the commands specified by the input source code. The interpreter maps its input, the source code, to an output, the executed operations. During the mapping process the interpreter needs to know the underlying syntactical structure of the source code. It uses a parser (also known as a syntax analyzer) that parses the input to determine the structure. [1]

Parsing can be thought of as transforming data from one representation to another. In the context of parsing programming languages, parsing would be the transformation of a source code from its textual human-readable representation to a syntactical representation. This syntactical representation usually is a Concrete Syntax Tree (CST) (also known as a parse tree) or an Abstract Syntax Tree (AST). The tools used to do these transformations are called parsers. [21]

Very often a parser is coupled together with a scanner (also known as a tokenizer or a lexer). The scanner's job is to go over the source code and recognize syntactical tokens. The tokens represent numbers, strings and other literals, keywords, identifiers, etc. The tokens are passed from the scanner to the parser, which uses them to create a parse tree.

The scanner must know what tokens to look for, and the parser must know what the syntactical structure is composed of. Both the tokens and the structure are defined using a grammar. A grammar is a set of terminal and non-terminal symbols, production rules that describe the syntax and a starting non-terminal (or a state). The production rules consist of a production head (a nonterminal) and a production body (a sequence of nonterminals and terminals). There are many formalisms for describing grammars. The most common one used for programming languages is the context-free grammar (CFG). [1]

There are many kinds of parsers. Among the most traditional ones are the LL and LR parsers and their modifications. Other ways of parsing include using parser combinators or parsing expression grammars (PEGs).

2.1 LL parsers

LL parsers process the input from left to right, and always perform the leftmost derivation. When we perform a derivation, we replace some non-terminal with a production rule body, where the head of the production rule is the non-terminal. LL parsers are top-down, this means that they build the parse tree from the root of the parse tree to its leaves. LL parsers are predictive, they look some amount of tokens ahead in the input and decide what production rule to perform. LL parsers are implemented via recursive functions, where each function represents a non-terminal. Each non-terminal function calls other non-terminal functions according to the production rules. The most common LL parser is the LL(1), which looks at most one token ahead in the input to decide which non-terminal function to call next. Because LL parsers are predictive parsers, the ambiguities we are concerned with are the FIRST-FIRST and FIRST-FOLLOW ambiguities. [1]

2.2 LR parsers

LR parsers process the input from left to right, and always perform the rightmost derivation. LR parsers employ a bottom-up strategy, they build the parse tree from the leaves to the root. They start with the terminals and perform either of two actions, shift or reduce, until there are no terminals left in the input and at the root of the parse tree is the starting non-terminal. The parser shift when it takes a terminal token from the input and puts it on a stack. The parser reduces when it replaces a sequence of both terminal and non-terminal symbols, that corresponds to a production rule body, on the stack with a non-terminal, that corresponds to the production rule head. To make a decision which action to perform, the parser can look at a token that is ahead in the input. Because the implementation of LR is very complex, they are almost never made by hand. Instead parser generators are used, which can generate the parser from a grammar description. [1]

Ambiguity in LR parsers comes in the form of shift/reduce and reduce/reduce conflicts. A shift/reduce conflict occurs when we are in a state where we can perform either a shift or a reduce action on the same lookahead token.

A reduce/reduce conflict occurs when the sequence of symbols on top of the stack matches multiple production rules' bodies.

LALR parsers are a modification of LR parsers. Compared to LR parsers, LALR parsers do not require as many internal parser states, which makes them more compact and easier to generate. [1]

GLR parsers are a modification of LR parsers, that allows shift/reduce and reduce/reduce conflicts to exist. If a conflict is encountered, the GLR parser forks the stack into multiple stacks and tries out all the possible actions in a breadth-first order. If some fork leads to a situation where no action can be taken, the GLR parser discards the entire fork. If more than one fork perform a successfull parse of the whole input, the GLR parser returns all the possible parse trees. [14]

2.3 Parser Combinators

Parser combinators are a functional way of constructing parsers. As the name suggests, parser combinators build parsers by combining less complex parsers together. This is usually achieved through higher order functions. One of the benefits is that parser combinators naturally mimic the structure of a grammar. That is beneficial for the developer, because he can design the grammar and implement the parser at the same time, using just one language. [20]

2.4 Scannerless Parsers

As the name suggests, scannerless parsers are parsers that are not coupled to a separate scanner. Instead the parser directly reads the input. The advantages of this approach [22]:

- No scanner. Eliminates the need to implement the scanner and the interface between the scanner and the parser.
- One common grammar for both lexical tokens and syntax.
- More expressive power when defining lexical tokens.
- Context guided scanner. Scanner does not try and match tokens that are not valid in a given context.

The disadvantages [22]:

- Worse maintanance. Lexical tokens are defined in the production rules of the grammar. This makes it less readable and the size of the grammar increases as well.
- Lower efficiency.

2.5 Parsing Expression Grammars

PEG is a formalism used to describe grammars. It may also be viewed as a way to describe a top-down parser. Its focus is on describing grammars of programing languages. Its advantage over a CFG grammar is the fact that it never introduces ambiguity. Instead of nondeterministic selection between alternatives in CFG, PEG is composed of ordered choices. [12]

2.6 Packrat Parsing

Packrat parsing is way of providing better parse time complexity to backtracking parsers, while making their memory complexity worse. With PEGs packrat parsing enables linear time parsing. The better time complexity is achieved through memoization, i.e. the parser remembers the result of every action that was previously executed at a given point in the input. [11]

Chapter 3

Pharo

Pharo is an open-source language and environment inspired by Smalltalk. It is fully object-oriented, e.g. primitives like numbers are objects, methods are objects and even classes are objects. It is a dynamically typed language. Pharo offers a live programming environment. You do not have to compile to see the changes in your code. You can inspect and modify anything in the environment while it is running. [2]

Pharo is cross-platform. The supported operating systems are: OS X, Windows, Linux, Android, iOS and Raspberry Pi. This is achieved via a virtual machine (VM) (which itself is written in Pharo). Thanks to this Pharo has found application in many diverse projects and fields including multimedia, educational and commercial. [5]

Pharo strives for small incremental updates and is driven by the support and feedback from the community. Everyone can have a say in Pharo's direction and success.[5]

3.1 Syntax

The syntax of Pharo is very, very simple. The syntax of the Smalltalk language from which it takes inspiration can fit on a single postcard [3] and so does Pharo's own syntax.

3.1.1 Syntactical Elements

In table 3.1 we can see almost all of Pharo's syntactical elements [5]. The only thing that is missing is the method syntax, which consist of the method name, arguments names and a body that consists of elements from the 3.1 table.

Syntactical element	Description
startPoint	a variable name
Transcript	a global variable name
self	pseudo-variable
1	integer
2r111	integer in radix 2 (binary)
1.5	floating point number
2.4e10	exponential notation
$a	a character `a` literal
'String'	a string
#Symbol	a symbol
#(1 2 3)	a literal array
{2 . 1 + 1 }	a dynamic array
"a comment"	a comment
|x y |	declaration of variables x and y
x := 1	assign 1 to x
[:x |x + 2]	a block closure with an argument `x`
⟨*primitive* : 1⟩	a virtual machine primitive or an annotation
3 factorial	unary message `factorial` sent to integer 3
3 + 4	a binary message `+` send
1 to: 3	a keyword message `to:` send
ˆ true	return the value true from a method
x printString . x := 4	the expression separator `.`
col add: 1; yourself	the message cascade operator `;`

Table 3.1: Showcase of Pharo syntactical elements

3.1.2 Message Sends

Everything in Pharo happens via sending messages, for example creating a subclass of a class happens by sending the `#subclass:` message to the class. Here are the three kinds of message sends ordered in precedence from highest to lowest:

1. unary message `5 asString`
2. binary message `1 - 1`
3. keyword message `Transcript show: 'Hello world!'`

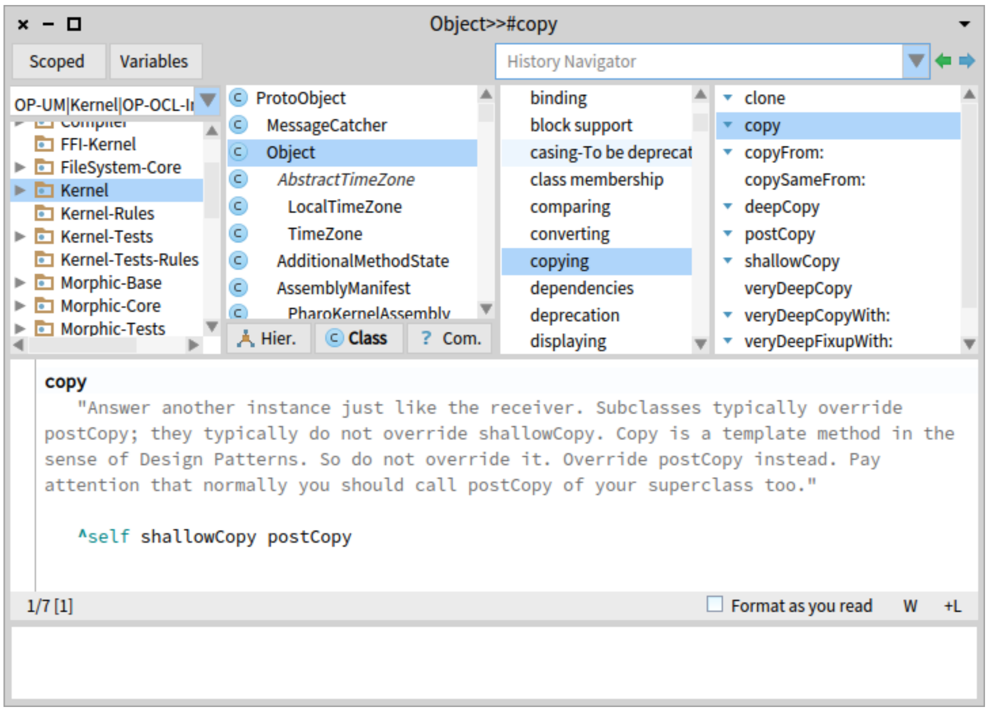
Message sends with the same precedence are evaluated from left to right. There are no other rules. It is necessary to understand that common arithmetic and logic operator precedence does not apply in Pharo, for example the expression `8 - 2 * 3` gets evaluated to 18.

3.2 Development Environment

The most useful tools that aid the development in Pharo are the: System Browser, Playground, Inspector, Debugger. [5]

3.2.1 System Browser

The System Browser can be opened by selecting it from the World Menu. The World Menu appears if you click on the Pharo window background.



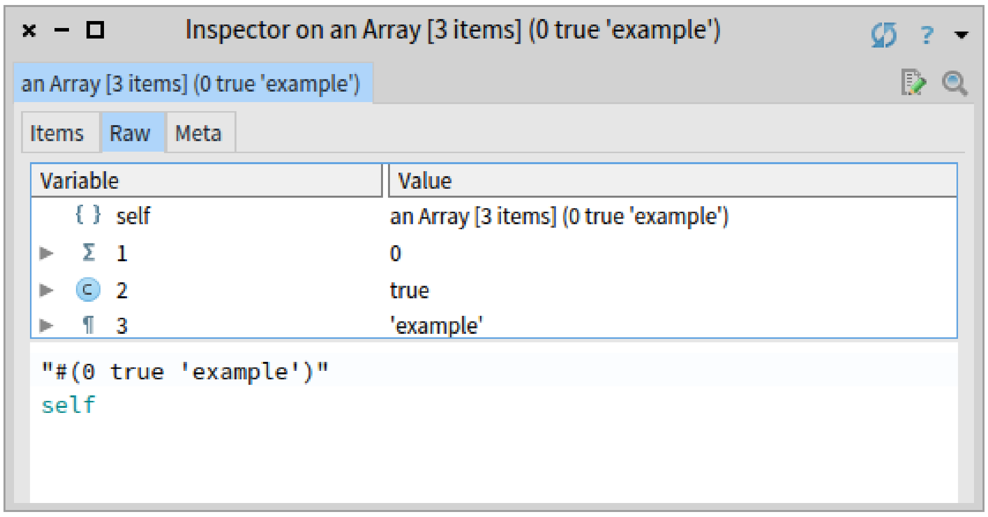
```
copy
	"Answer another instance just like the receiver. Subclasses typically override
postCopy; they typically do not override shallowCopy. Copy is a template method in the
sense of Design Patterns. So do not override it. Override postCopy instead. Pay
attention that normally you should call postCopy of your superclass too."

	^self shallowCopy postCopy
```

Figure 3.1: The System Browser

The System Browser is where most of the development process takes place. In figure we can see the System Browser. The System Browser is made up of several panes. The top four panes are in order from left to right: the package pane, the class pane, the protocols pane and the methods pane. The pane in the middle serves to implement a method or to declare a class. The pane in the bottom is for warnings and suggestions.

3.2.2 Inspector

The Inspector is a window that shows the details of an object, like attribute values, and allows to evaluate Pharo code on the object. It can be opened by evaluating Pharo code with `Ctrl/Alt + i` or by selecting the Pharo code, right-clicking and selecting `Inspect it` from the popup menu.


Inspector on an Array [3 items] (0 true 'example')
an Array [3 items] (0 true 'example')
Items
Raw
Meta

Variable	Value
{ } self	an Array [3 items] (0 true 'example')
Σ 1	0
2	true
¶ 3	'example'

```
"#(0 true 'example')"
self
```

Figure 3.2: The Inspector

Chapter **4**

Pharo's parser frameworks

There are not many parsing frameworks avaiable for Pharo. The only two robust and feature-full frameworks are the PetitParser and SmaCC parsing frameworks.

4.1 PetitParser

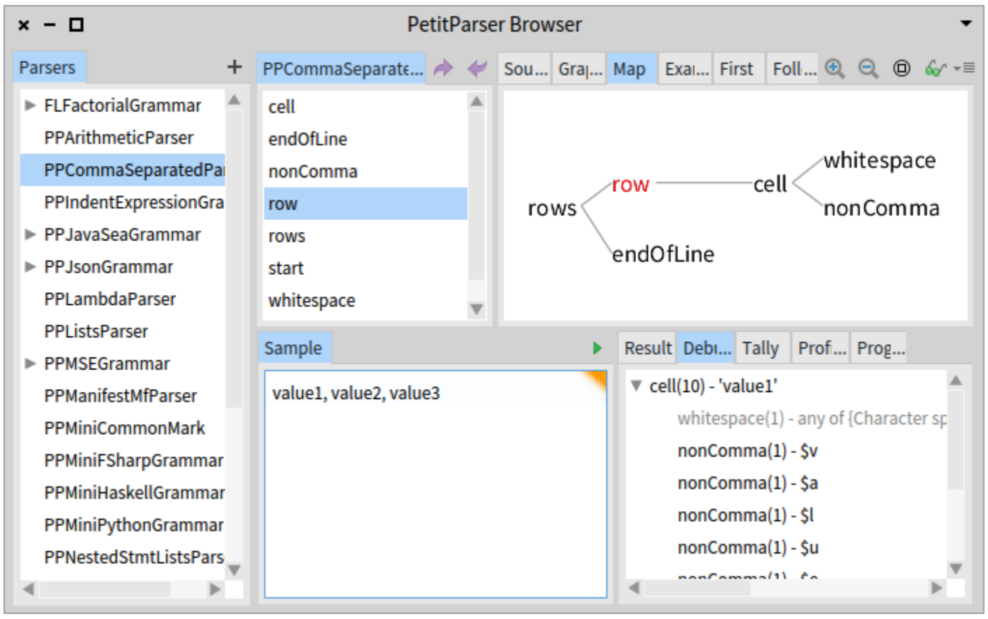
PetitParser uses a combination of scannerless parsing, parser combinators, PEGs and packrat parsers. This combination of different parsing methodologies is what makes PetitParser expressive in what languages it can parse. The benefits of using PetitParser is that we can write the grammar using Pharo code. That makes it easier to debug, modify and extend and fits more naturally into the Pharo environment. [4]

In listing 4.1 we can see a definition of a parser that parses and evaluates an expression made out of numbers and additions.

```
|num exp|

num:= #digit asParser plus flatten ==> [:str | str asNumber].

exp := ( num , $+ asParser trim , num ==> [ :nodes |
       nodes first + nodes last
]) / num.

exp parse: '4 + 3 + 2'    " parses and evaluates to 9"
```

Listing 4.1: Example of PetitParser grammar fragment

PetitParser comes with a PetitParser Browser tool that integrates grammar visualization, development and debugging in one. We can see it in figure 4.1. In the left pane we have a list of all PetitParser parsers avaiable in the image. In the top middle-right part of the browser we can see the list of parser methods that each represent a production rule in a grammar. Then we have features like grammar tree visualization, looking at the first and follow sets. The bottom middle-right part is saved to evaluation language snippets and debugging.



```
value1, value2, value3
```

```
▼ cell(10) - 'value1'
    whitespace(1) - any of {Character sp
    nonComma(1) - $v
    nonComma(1) - $a
    nonComma(1) - $l
    nonComma(1) - $u
```

Figure 4.1: The PetitParser Browser

4.2 SmaCC

SmaCC is a LR parser generator originally for Smalltalk created by John Brant and Don Roberts for the VisualWorks 7, Dolphin 6.0 Professional, and VASmalltalk 8.6.1 platforms. SmaCC is able to parse ambiguous grammars, left and right recursion, and overlapping scanner tokens. It is also capable of GLR parsing as well. The creators of SmaCC used it to write many migration and code tranformation tools, e.g. they used SmaCC to migrate 1.5 million lines of code from Delphi to C#. [6]

SmaCC was ported to Pharo and is actively mainted by Thierry Goubier. It also received several updates and new features, like the capability to generate LALR parsers and a better debugger.[13]

4.2.1 SmaCC Parser Generator

SmaCC provides an UI tool to write down and develop our grammar, we can see it in figure 4.2. It can be opened from the `World Menu` → `Tools` → `SmaCC Parser Generator`. The first of the three fields at the top serve to input our package name, where the parser and scanner will be generated. The second and third one are the names of our scanner and parser, respectively. Under the three fields, there are buttons for saving, loading, reverting to a previous version and generating either a LR parser or a LALR one. Under the buttons is the main text field where we input all of our grammar together with parser declarations and token definitions. There are four tabs under the text area. These provide useful features for testing and debugging our grammar. The messages tab prints SmaCC warnings, including AST generation warnings and ambiguity warnings. Item sets tab corresponds to the LR item sets. The symbol tab prints all used terminals and non-terminals. The test tab is where we can input a string and parse it with our parser either inspecting the parsed result or opening a modified debugger that can go through the whole parsing process.

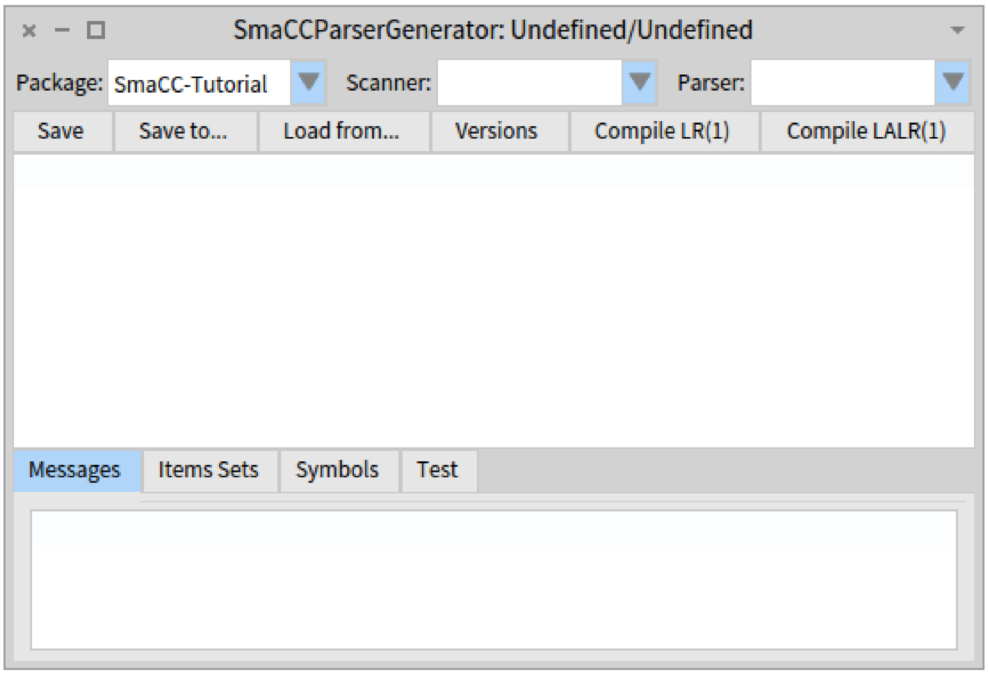


Figure 4.2: The SmaCC Parser Generator

4.2.2 SmaCC Syntax

4.2.2.1 Scanner Tokens

Scanner tokens must be defined before they are used for the first time. The token definition syntax is:

```
"<" TokenName ">" ":" Definition ";"
```

Example: `<IfKeyword>: if;` defines an if token named IfKeyword.

In the Definition part of the syntax we can use regular expressions. All the possible regular expressions are defined in [13].

Example: `<Number>: ( 0 | [1-9][0-9]* );`

In our grammar rules we can also define scanner tokens by putting a string in double quotes. Regular expressions do not work in the double quotes.

Example: `"else"` defines an else scanner token .

4.2.2.2 Grammar Rules

Grammar rules consist of non-terminals and their production rules which consist of other non-terminals and scanner tokens. The syntax is:

```
NonTerminal ":" ProductionRule ( "|" OtherProductionRule )* ";"
```

Example: `Expression: <Number> "+" <Number> | <Number>;` a parser with this rule could parse `'21 + 21'` or `'42'` substring .

Without these rules being annotated by actions or parse tree declarations, the parser will return a collection of `SmaCCTokens`. Each `SmaCCToken` contains information about the string value of the parsed substring they correspond to. Additionaly they contain the substring start and end position, and an internal scanner token id.

4.2.2.3 Actions

Actions allow us to do something with the parsed result, i.e. allow us to do something with the collection of `SmaCCTokens`. We can see an example in listing 4.2. Inside the action braces we can use regular Pharo code. We named the tokens and non-terminals so that we can reference them in the Pharo code. We could also reference them with a string containing a number depending on the order they appear in the production rule, but that is not a good style. It mainly causes problems when we change the rule later.

```
Expression
    : Number 'left' "+" Number 'right' {left + right}
    | Number 'num' {num}
    ;

Number
    : <Number> 'tok' {tok value asNumber}
    ;
```

Listing 4.2: Actions example

The result of parsing a string `'21 + 21'` would now be the number 42.

4.2.2.4 Directives

There is a large number of SmaCC directives and we can find the whole list in [13]. We will list few important ones:

- `%root Name;` sets the root of the parse tree class hierarchy to Name. The Name can be any existing non-terminal or a new name and in that case SmaCC will generate a new class for it. We will look at Parse trees in 4.2.2.5.
- `%left ScannerToken;` important tool with dealing with shift/reduce conflicts, tells the parser to reduce if it encounters the ScannerToken token.
- `%right ScannerToken;` same as `%left`, but the parser will perform a shift
- `%glr;` tells the compiler to generate a GLR parser.
- `%unicode;` tells the compiler to generate a scanner that can handle unicode strings.
- `%start NonTerminal (NonTerminal)*;` tells the compiler to generate starting states for the non-terminals. Useful for debugging or incremental grammar development.
- `%hierarchy NonTerminal "(" ListOfNonTerminals ")";` the non-terminals in the ListOfNonTerminals are going to be generated as subtypes of the NonTerminal.

4.2.2.5 Parse Tree

Instead of actions we can annotate production rules with Parse tree node declaration. For the tree node declarations to work we have to have the compiler directive `%root` set. In listing 4.3 we removed the action braces and replaced them with double braces. We can put a name in the double braces, which will then be used as a base for the generated parse tree class. If the double braces are empty it means the generated class name base will be the name of the corresponding non-terminal. The generated tree hierarchy will be `ParseRootNode`, `ParseAdditionNode` and `ParseNumberNode`. The ParseAdditionNode will have two instance variable: `left` and `right`. Smacc also generates a tree visitor for the generated parse tree class hierarchy and an accept method for each of the tree classes.

```
%root Root;
%prefix Parse;
%suffix Node;

<Number>: ( 0 | [1-9][0-9]* );

Expression
    : Number 'left' "+" Number 'right' {{Addition}}
    | Number 'num'
    ;

Number
    : <Number> 'tok' {{}}
    ;
```

Listing 4.3: Actions example

Part II

Practical Part

Chapter 5

Analysis and Design

To implement an OCL interpreter we need several things: a grammar that describes the language, a scanner (we do not need a scanner with scannerless parsers), a parser and an implementation of the standard library.

The OCL specification[16] offers a complete grammar in the EBNF format. My intention at first was to use this grammar as there are guarantees that it can be used to generate the language correctly, since it is provided through official channels. It would also be great for future development. If somebody else would start working on the interpreter, he or she would find most of the needed information in the specification.

5.1 Using an LL parser

Since I already had some experience with creating LL parsers, I was looking into the possibility of creating a handmade LL parser. This would have the benefits of being completely familiar with the whole system and since LL parsers are relatively simple to implement it would not be such a problem. Nevertheless I soon realized that this will not be an option, because the grammar contains many ambiguities and left-recursion. The transformations that would have to be done to get rid of these ambiguities would be managable, but the grammar would change to the point where the original motivation for using the official grammar would be gone. What is even a bigger problem is the left-recursion. The commonly used algorithms for removing left-recursion in the worst case increase the amount of production rules and symbols exponentially[15]. This would make the resulting grammar almost impossible to implement and maintain.

5.2 Using PetitParser

There were two options at this point, either write my own grammar or look for some tool or framework that can handle both left-recursion and ambiguity. There were only two avaiable parsing frameworks for Pharo, PetitParser and SmaCC. I chose PetitParser at first, because it appealed to me with the fact that you write down the grammar using Pharo code. That would mean that there would not be any delay caused by grammar compilation as it is with SmaCC. And because we could utilize the entirity of Pharo, any semantics check or input sanitization could be done directly in the parser without the need to do it later outside of the parser.

There is just one problem. PetitParser cannot actually deal with left-recursion. What made me think that it could and waste time is this line in [4]: “Packrat Parsers give linear parse-time guarantees and avoid common problems with left-recursion in PEGs”. Since PEGs cannot deal with left-recursion at all and there exist Packrat parser modifications that allow it to deal with left-recursion[23], this made me believe that PetitParser can actually deal with left-recursion. What the the sentence in the quote was actually refering to is the PEGs’ problem with backtracking, which could make the parsing time proportionaly exponential to the input size. Packrat parsers deal with the backtracking problem through memoization. Since [4] is one of the very few full-length sources for PetitParser in Pharo, I deemed neccassary to warn against the poor choice of words in the quoted sentence.

5.3 Using SmaCC

Since SmaCC generates LR or LALR parsers it can deal with left-recursion. Ofcourse with LR and LALR there are different kinds of ambiguities then with top-down parsers. With shift/reduce ambiguities SmaCC deals using the `%left` and `%right` directives which tell the parser whether it should shift or reduce on certain tokens (reduce for `%left`, shift for `%right`). By default SmaCC shifts. With the use of the SmaCC directive `%glr` we can turn on GLR parsing and deal with reduce/reduce ambiguities. With GLR parsing SmaCC tries all possible parses. If more parses are valid SmaCC raises a `SmaCCAmbiguousResultNotification` exception which can be caught and the ambiguity dealt with.

This means that I could use the official OCL grammar. The idea was to rewrite the whole grammar from the specification’s format to the SmaCC’s format. Both of the formats are very similiar, but it still was a lot of work, mainly because of naming all the important symbols in the production rules, creating appropriate parse tree classes, dealing with SmaCC quirks (e.g. SmaCC not being able to handle unicode characters with values over FFFF, grammar idioms that resulted in changing up some of the rules, etc.).

My plan to deal with the ambiguities was to have the parser try and parse the whole input and return all possible parse trees . Since SmaCC provides a parse tree visitor class, I wanted to subclass this visitor and walk all the parsed trees and invalidating those that would not be correct. To decide which tree was not valid we would need context information, like what types, variables or implicit attributes are avaiable. To give an example of an ambiguity, if we parse the string `'identifier'` we do not know whether it is a variable, an attribute of an implicit variable, or if it is a type. SmaCC by default throws the ambiguity exception as soon as it can. This is a problem, because then I do not have the information about variable declarations, implicit/explicit iterators or context definition declarations. To resolve this issue I emailed the author of Pharo's SmaCC port, Thierry Goubier. We exchanged few emails and he suggested several possible solutions, among them one was exactly what I needed. To force SmaCC to parse the input all the way we need to initiate the parsing through `parseAll:startingAt:` method, which takes the input string and the starting state that corresponds to a non-terminal in the grammar. Altough it worked for smaller OCL snippets of code, when I tried to parse the context invariant discussed in subsection 7.2.2 the whole Pharo image froze. I underestimated the quantity of ambiguities present in the official OCL grammar.

I emailed Thierry Goubier once more and he mentioned that he is, together with his student, working on improving the handling of ambiguities. Their plan is to, instead of throwing the ambiguity exceptions, program the parser to insert ambiguity nodes, which would contain the different parse possibilities, into the parsed tree which could then be subsequentially handled by a visitor. This would solve my problem as there would be only one tree. But this feature is currently only in development.

5.4 Final solution

At the end I decided to let go of my idea to use the official OCL grammar, instead I would write my own grammar.

Because I lost a lot of time with all my previous attempts I was forced to limit what I could support in the final interpreter. I had to support everything that would be needed for the final OntoUML usecase (discussed in section 7.2). That includes:

- context invariant syntax
- `SimpleName` for variables, types, implicit attribute navigations

- `SimpleName(Arguments)` for implicit object operation calls
- `OCLExpression.SimpleName` for explicit attribute navigations
- `OCLExpression.SimpleName(Arguments)` for explicit object operation calls
- `OCLExpression->SimpleName(Arguments)` for collection operation calls
- `OCLExpression->IterationName(VariableDeclarations | Arguments)` for collection iterator expressions
- Let expression syntax, for helper variables
- If expression syntax
- Unary and Binary operator syntax
- Primitive and Type Literals
- Variable declaration syntax
- Arguments list syntax

This is the extent of the syntax the interpreter supports. What it does not support:

- most of the context declarations: context definitions, derivation and init values, operation pre- and postconditions, operation body
- messages and signals
- `@pre` to access pre-operation call attribute values in postconditions
- `SimpleName[index]` and `OCLExpression.SimpleName[index]` for attributes that are of a collection type and associations
- Enum, collection, null and invalid literals
- Path names for namespace specification
- Iterate expression
- Collection shorthands

Extending the interpreter to support these expressions is just a matter of writing the syntax, ensuring it is not ambiguous and creating the appropriate AST classes.

The whole syntax is shown in appendix A. It is worth mentioning that the grammar does contain ambiguities. It does not contain any shift/reduce conflict that could not be taken care of through a SmaCC directive. It also contains reduce/reduce conflicts, but there is always only one valid parse tree that the GLR can return. I always made sure that this is true using the SmaCC Parser Generator's Messages tab, which prints any shift/reduce and reduce/reduce conflicts. In the current state of the grammar the Messages tab is empty. It is always a good idea to check this tab when extending the grammar as there were numerous times where I was certain that my grammar extension does not introduce any ambiguity only to be proven wrong by a warning message.

Now for the AST. The AST hierarchy is inspired by the OCL specification. The two main differences are:

- the implemented AST does not support every type that is specified and it does not support every operation on the types that are supported.
- the AST hierarchy in the specification is at some point integrated into the UML hierarchy, now this was not possible for me to do. Most of my time on this thesis was spent digging through the 262 pages long OCL specification to understand it enough so I could implement the interpreter. The idea that I would go through the official UML specification[17] that is almost 800 pages long was out of the scope of this thesis. It would be more suitable for someone who already has a deep understanding of the UML specification or maybe it would fit a master's thesis scope.

Again I included just what was necessary for the OntoUML constraint.

Since the OntoUML constraint is a constraint expressed on the metamodel, I only included the ability to interpret metaOCL constraints on a instance of a `OPUMLModel`. I believe that to extend the interpreter to handle domain-specific cosntraints, one would have to solve the problem with integrating the OCL AST hierarchy into UML, since to find all the constraints defined on the model elements, to lookup all instances and to type-check every expression, etc., one would need types, attributes and values that are defined in the UML model itself.

Integration into OpenPonk was done by adding a `metaOCL` instance variable to the `OPUMLMetaElement` class and adding a `gtInspectorMetaOCLIn:` method that extends the inspector opened on any `OPUMLMetaElement` instance including the instance of `OPUMLModel` by adding a text field where you can input the OCL constraint source code and save it either via a button or by pressing `Ctrl/Alt+s`. The control character (Ctrl or Alt) depends on the operating system that is used to run Pharo. You can then evaluate the OCL source code by calling the `OPOCLInterpreter` method `evaluateMetaOCLOnModel:` on the model instance. The most convenient place to do this is in the inspector's Raw tab, where you can evaluate Pharo code.

As a final note of this chapter I would like to mention that it would be possible to write a grammar that would not be left recursive and without ambiguities, that would be managable and therefore it would be possible to implement an LL parser for it by hand or use PetitParser. But since I was constrained by time at this point, it did not seem viable to spend more time to write a more complicated grammar.

Chapter 6

Implementation

In this chapter we will look at the final state of the implementation. We will look at the packages, important classes and the interpretation process. Since all the classes are commented in the code and I kept most of the methods small and self-explanatory, I will just provide a short description of each package and its most important classes.

During the implementation of the interpreter I used:

- **Pharo** version: 6.1
- **SmaCC** version: not avaiable
- **OpenPonk** version: master 105

All the classes and packages are in the `OP-OCL-Interpreter` package. The package does not actually contain other packages, instead it is subdivided using Pharo tags, but I will refer to these as packages as well. The packages are:

- `OP-OCL-Interpreter-Core`
- `OP-OCL-Interpreter-CST`
- `OP-OCL-Interpreter-AST`
- `OP-OCL-Interpreter-AST-Expressions`
- `OP-OCL-Interpreter-AST-Types`
- `OP-OCL-Interpreter-AST-Values`
- `OP-OCL-Interpreter-ErrorHandling`
- `OP-OCL-Interpreter-Testing`

I will be referring to the packages without the `OP-OCL-Interpreter-` prefix.

6.1 Core Package

The core package only contains the class representing the OCL interpreter.

6.1.1 `OPOCLIntepreter`

Class representing the OCL interpreter. This is the main class in the implementation. It drives the whole interpreting process.

Instance variables:

- `environment` = Instance of `OPOCLEnvironment`. The interpreter uses it to store instances of and references to classes from the `AST-Types` package. The interpreter also manages the OCL `self` context variable. This environment is then passed to an instance of `OPOCLASTBuilder` which uses the types in the environment for expression type conformance.
- `model` = Reference to the `OPUMLModel` instance that the interpreter was called on.
- `instances` = An instance of `Dictionary`. Keeps track of all the instances in the model to be later used in context declarations. In the current implementation just stores the model.

An important method is the `evaluateMetaOCLOnModel:` which takes a model instance as an argument an evaluates the OCL constraint that is defined in the model.

6.2 CST Package

The `CST` package contains classes related to parsing and the CST. It contains the entire hierarchy of classes that represent nodes in the CST, visitor classes for the CST nodes. It has the class that represents the environment and also the OCL parser and scanner classes. Except the `OPOCLASTBuilder` and `OPOCLEnvironment` classes, most of the classes in this package are auto-generated with only minor modifications. They are auto-generated by SmaCC based on the handwritten grammar described in appendix A.

6.2.1 `OPOCLParser`

This class represents the OCL parser. This is a GLR, LALR parser.

The parser is used via the `parse:` method. It takes a string argument, that represents the OCL source code. By default the starting state of this method is the topmost production rule's head non-terminal. In this case it is the ContextInvariant non-terminal.

Handmade modifications to the parser:

- It is important to extend the instance side `tryAllTokens` method and return `true` so that when the scanner matches the next input to more tokens, our parser tries them all.
- Added multiple class side `parse*` methods for testing purposes, e.g. `parseExpression:`, `parseVariableDeclaration:`. These behave as the `parse:` method with the starting state set to a different non-terminal.

6.2.2 `OPOCLEnvironment`

Responsible for keeping track of all the avaiable types and variables in a given scope. An environment can have another parent environment. This is used to create nested variable scopes. All the variables, type, methods and attribute lookup methods work recursively. First we check in the current environment(scope) and then try to perform the lookup on the parent environment.

6.2.3 `OPOCLASTBuilder`

This class inherits from the auto-generated `OPOCLNodeVisitor`. Responsible for traversing the CST tree and constructing a corresponding AST tree. During the construction of the AST, the builder has to lookup variables and types in its `OPOCLEnvironment currentEnvironment` instance variable and perform type conformance checks. After setting the builder's environment it can be called on a instance of `OPOCLCSTNode` via the `accept:` method.

6.2.4 `OPOCLCSTNode`

The root of the CST class hierarchy. Represents a node in the CST. Every node has an `ast` instance variable that represents the corresponding AST node. This variable is set by the `OPOCLASTBuilder`.

6.3 AST Package

The `AST` package contains classes that did not fit into any of the other AST packages. The classes in the `AST*` packages, in general, are classes that either represent nodes directly in the AST, classes that are used by these nodes, or visitor classes.

6.3.1 `OPOCLAttribute`

Represents an attribute of `OPOCLType` classes. The attribute has a string `identifier` and a `type`. The `type` is an instance of the `OPOCLTypeIdentifier` class. Used by `OPOCLType`, `OPOCLEnvironment` and `OPOCLASTBuilder`.

6.3.2 `OPOCLContextInvariant`

Represents the OCL context invariant. Instance variables:

- `contextName` = A string representing the context in which the invariant is declared.
- `invariantConstraint` = An instance of `OPOCLOCLExpression`. This expression is evaluated by the interpreter to check if the constraint is valid.
- `invariantName` = An optional string representing the invariant name. Used only in user feedback, e.g. error messages.

6.3.3 `OPOCLOperation`

Represents an operation signature. Used in `OPOCLType` classes to describe the methods the particuliar type can perform. Instance variables:

- `identifier` = Instance of a string. Represents the method's name.
- `returnValueTypeIdentifier` = The `OPOCLTypeIdentifier` of the return value. Used in type conformance checks when building the AST.
- `argumentsTypeIdentifier` = An array of `OPOCLTypeIdentifier` representing the operation's arguments.

6.3.4 `OPOCLTypeIdentifier`

Represents a type identifier. The reason why this class exists, instead of using a string, is because of collection types, which have nested element types. I could neither use an instance of an `OPOCLType`, because the concrete types in the interpreter get instantiated right before the interpreting process begins. This is because each instance of the `OPOCLInterpreter` has to have its own types, since OCL allows to modify even the standard library types through the context definition declaration.

6.4 AST-Expressions Package

Except the `OPOCLExpressionVisitor` classes, the classes in this package represent the OCL expressions described in 1.1.

6.4.1 `OPOCLExpressionEvaluator`

This class visits the expression tree and computes the value of the expression nodes. It also keeps track of the avaible variable values in its `variableInstances` instance variable. The evaluator does not do any type checking or method avaiability. This is all done in the `OPOCLASTBuilder`.

6.4.2 `OPOCLOCLExpression`

Represents an OCL expression. The root of the expressions hierarchy. Each expression has a `type` instance variable. The `type` is an instance of an `OPOCLType` class, not the `OPOCLTypeIdentifier`. The `type` is set by the `OPOCLASTBuilder`.

The expressions hierarchy mimics the expressions hierarchy in the official OCL specification [16].

6.5 AST-Types Package

Classes in this package represent the type hierarchy. As with the expressions, the types hierarchy mimics the types hierarchy in the official OCL specification [16].

6.5.1 OPOCLType

The types are used throughout all the main classes. They describe type names, attributes and methods. They are used by: `OPOCLOCLExpression`, `OPOCLInterpreter`, `OPOCLASTBuilder`, `OPOCLEnviroment`, `OPOCLValue` and `OPOCLExpressionEvaluator` classes. Instance variables:

- `oclName` = Instance of a string. Represents the type name in OCL, e.g. `OPOCLBooleanType`'s `oclName` is `Boolean`.
- `attributes` = A dictionary of `OPOCLAttribute` instances. Represents the type's attributes.
- `operations` = A collection of `OPOCLOperation` instances. Represents the type's operations.

6.5.2 OPOCLCollectionType

Collection types additionally have an `elementType` instance variable that represents the `OPOCLType` of the collection elements.

Collection types are also used differently by the `OPOCLInterpreter`, `OPOCLEnvironment` and `OPOCLASTBuilder`. Thanks to the `elementType` there is an infinite number of different collection types. That is why the interpreter stores the collection type classes, and not their instances, in its `metatypes` dictionary. When the `OPOCLASTBuilder` encounters a collection type that has not yet been instantiated but its metatype is present in the environment a new collection type is instantiated and added.

Example: The `OPOCLASTBuilder` encounters a `Bag(Integer)`, it looks up the `Bag(Integer)` in its `currentEnvironment`. The environment does not have an instance of the `Bag(Integer)` type yet, but it has the `Bag` metatype stored, so the environment automatically instantiates a new `Bag(Integer)` collection type and returns it to the builder. The `OPOCLASTBuilder` can then continue.

6.6 AST-Values Package

Classes in this package represent expression, instance and variable values.

6.6.1 `OPOCLValue`

Represents the values that are either computed from expressions, or belong to the model element instances or variables. Each `OPOCLValue` class has a corresponding `OPOCLType` class. Whereas the `OPOCLType` classes contain a description of what attributes and methods the type has avaiable, the `OPOCLValue` classes actually implement these methods and attribute getters/setters. Instance variables:

- `instanceValue` = A reference to some Pharo object. The implemented methods then manipulate this object. In the case of collection values, their `instanceValue` is a Pharo collection object, whose elements are instances of a `OPOCLValue`. This is important for expressions where the collection's elements are accessed, e.g. in an iterator expression.
- `typeInstance` = An instance of a `OPOCLType`. Represents the value's type.

Some of the `OPOCLValue` classes are basically mapping classes, e.g. the `OPOCLModelValue` maps to the OpenPonk's `OPUMLModel`. That means that for example the `OPOCLModelValue`'s `packagedElements` method does just this: `^ instanceValue packagedElements`.

6.7 ErrorHandling Package

The `ErrorHandling` package contains all the specialized exceptions thrown during the interpreting process. There is also a general purpose visitor avaiable to be subclassed for better error message feedback. The description of what exceptions and under what circumstances they are signaled, is avaiable throughout the section 6.9.

6.8 Testing Package

As the name suggests this package contains the tester classes. The list of the test classes and what they test:

- `OPOCLParserTest`: Since the methods in the parser are auto-generated and complicated, this class tests the grammar rules itself. I decided to prioritize this class in test coverage. It covers all of the production rules defined in the grammar.
- `OPOCLEnvironmentTest`: Tests the environment's add, lookup and nested environment creation methods.
- `OPOCLASTBuilderTest`: Tests the functionality of the `OPOCLASTBuilder` class. It tries to minimize dependency by creating `SmaCCToken` instances and other CST classes by hand. Only the `visitInteger:` and `visitImplicitNavigation:` methods are tested.
- `OPOCLValueTest`: Tests the `OPOCLValue oclIsKindOf:` method.
- `OPOCLBooleanValueTest`: Tests the `OPOCLBooleanValue or:` method.
- `OPOCLExpressionEvaluatorTest`: Tests the `OPOCLBooleanValue or:` method.

Some further comments about the tests are in subsection 7.1.4.

6.9 Interpreting

In figure 6.1 we can see an overview of the whole interpreting process. When the `OPOCLInterpreter` interpreter is called to evaluate OCL on a metamodel, it first loads its types. This means that the interpreter creates instances of the `OPOCLType` classes and stores them in its `OPOCLEnvironment environment` instance variable.

The interpreter then loads its metatypes. Contrary to loading types, the interpreter stores the `OPOCLCollectionType` classes themselves, not their instances. It will instantiate and store instances of the `OPOCLCollectionType` classes as is needed during evaluation. If the interpreter were to be extended, this would be done for the tuple and template types as well.

The last step before processing the OCL constraint, is loading instances into the `Dictionary instances` instance variable. In the current implementation this just means storing the `OPUMLModel` instance, on which the interpreter is being called. With a more complete implementation the parser would store all instances of classifiers, associations, stereotypes, etc. for metamodel constraints. In domain specific constraints the interpreter would store instances of the types defined in the model itself, this would require UML object diagrams.

Next step is the parsing of the input and creating a CST. This is done by calling the `OPOCLParser parse:` method that takes in a string, the source code for the OCL constraint. If the source code is syntactically valid it returns the CST root, otherwise it signals a `SmaCCParserError`.

The CST root is then passed to an instance of `OPOCLASTBuilder` which visits every node of the CST. The builder performs disambiguating actions, complete type conformance checks and creates corresponding AST nodes. During this process new collection types are instantiated as they are encountered in the tree. Examples of the errors the builder can encounter are: all expressions do not have valid type conformance, a variable or a type is being referenced that does not exist, trying to declare a variable with a name that is already in the scope. Two exceptions can be signaled: `OPOCLEnvironmentError` or `OPOCLASTBuilderError`.

The interpreter takes the context invariant (in the current implementation, you can only define one invariant on a model, this is a shortcoming that could be amended by extending the grammar's context invariant production rule and modifying the logic in the interpreter) and passes its invariant expression to an instance of `OPOCLExpressionEvaluator`. The evaluator goes over all the expressions in the AST and for each node returns an instance of the `OPOCLValue` classes. There are three distinct ways that the evaluator calculates the values. When the evaluator visits a literal expression, it just returns the value of the literal, which was created during the AST construction. When the evaluator visits a variable expression, the evaluator looks for the value in its `variableInstances` dictionary. Upon visiting the other types of expressions, the evaluator computes the value by sending `perform:with:` messages to the objects in the expression. The `perform:with:` is the Pharo's `Object` method that takes a string, which represents the method identifier, and as a second argument takes an object. In this case it takes an array of arguments. A special case is the iterator expression, where the evaluator sends a `performIteration: withPrimaryIterator: withSecondaryIterator: withBody: usingEvaluator:` message. This is the only case where a value type has to do more than just passing a message to its `instanceValue`. I was not able to design a cleaner solution. The problem is that the return value type and thus the way the iteration is performed, and how many iterators the iteration needs, depends on the specific type of the value and the iteration, but since the evaluator is the only thing that can evaluate expressions, the value needs a reference to the evaluator and use it to evaluate the iteration body expression. When performing the iteration the value has to check for the correct number of passed iterators. If the number of iterators is invalid, the value signals an `OPOCLEvaluationError`.

After the evaluator has finished evaluating the invariant, the interpreter checks whether the value is true. In that case the interpreter returns true to the original caller. If the value is false then that means that the invariant is invalid for the model instance. In that case the interpreter signals an `OPOCLEvaluationError`, stating the invalid invariant name and the context name of the invariant.

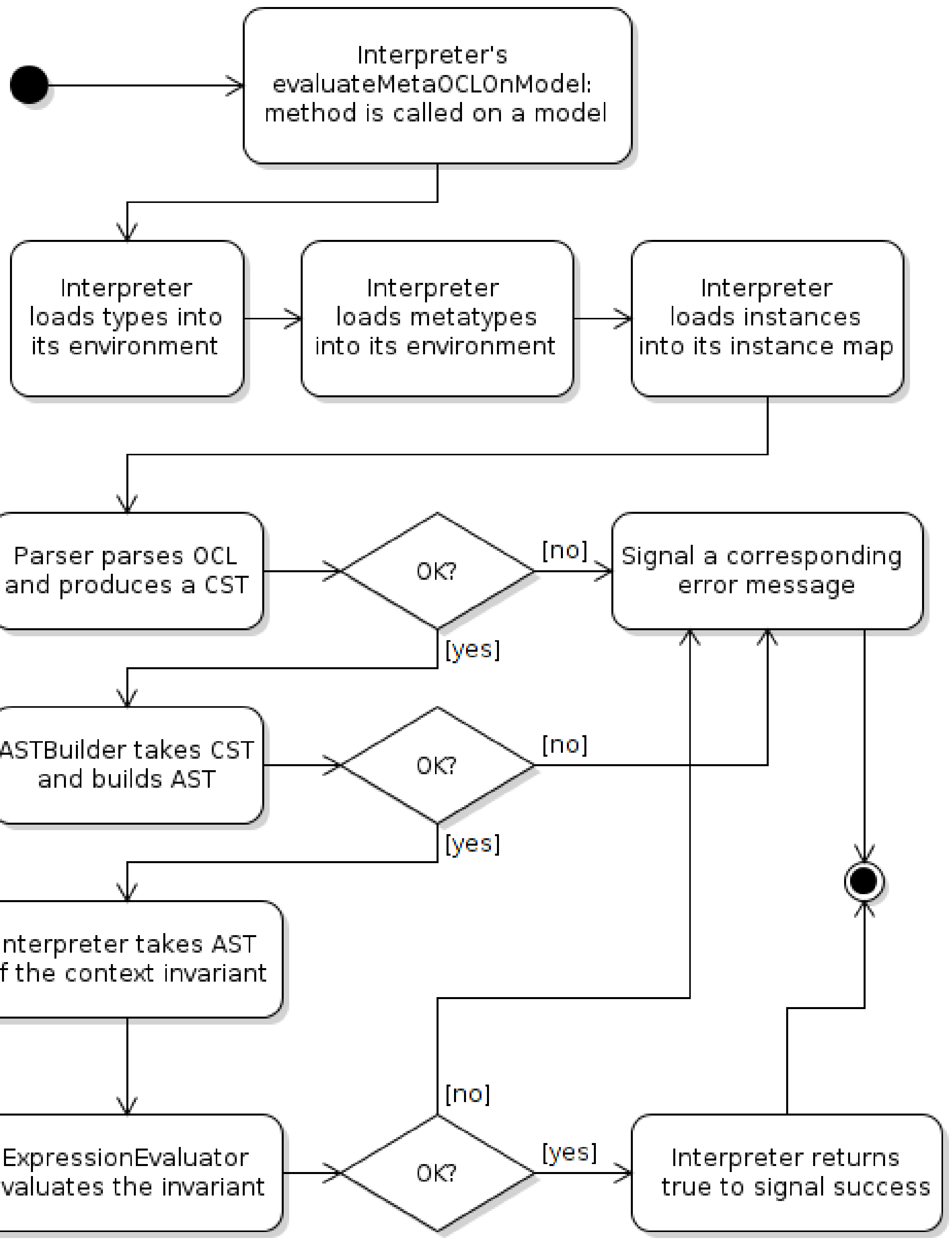

Interpreter's evaluateMetaOCLOnModel: method is called on a model
Interpreter loads types into its environment
Interpreter loads metatypes into its environment
Interpreter loads instances into its instance map
Parser parses OCL and produces a CST
OK?
[no]
Signal a corresponding error message
[yes]
ASTBuilder takes CST and builds AST
OK?
[no]
[yes]
Interpreter takes AST of the context invariant
ExpressionEvaluator evaluates the invariant
[no]
OK?
[yes]
Interpreter returns true to signal success

Figure 6.1: The activity diagram of the OCL interpretation process

Chapter 7

Testing

In the first section of this chapter we will look at what kind of automated tests were written during the implementation process and what tool was used to perform these tests. In the second section we will look at a use case and showcase the OCL interpreter in action.

7.1 Unit and Integration Tests

During the development and maintanance of a software product it is very beneficial to have a set of automized tests. It helps the developer with modifying and/or extending the current codebase. Whenever a modification is made the developer can run these automated tests and if they all pass, the developer knows that he did not break any other part of the application. And in the case that something actually breaks the developer at least knows about it immiedietly.

7.1.1 Unit Tests

In [18] Roy Osherove says that unit testing is way of writing tests that has been here since the beginning of Smalltalk, that is for almost 50 years. For Osherove a unit test has the following traits:

- It is automated
- It tests one ”unit of work”, which is a method for example
- It asserts the result of the unit of work
- It is almost always written using a framework
- It should take almost no time to execute
- It should be readable, maintable and the result of the assertion should not change unless the codebase does

The strength of unit tests lies in the fact that they test only one functionality (a method) of the system and have little to no dependency on the rest of the system. Unit tests should be run as frequently as possible.

7.1.2 Integration Tests

Integration tests contrary to unit tests test more than one thing and do not care for dependency. Their purpose is to test the interactions between different parts of the application. Integration tests should be executed on pieces of code that were already unit tested.[19]

7.1.3 SUnit

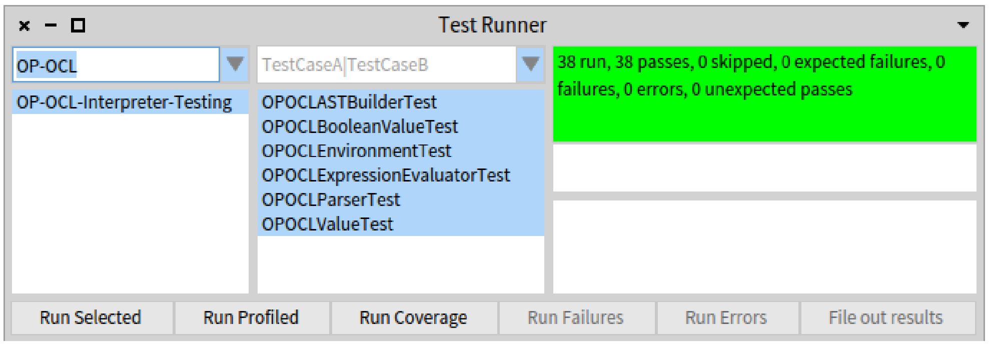
SUnit is the main Pharo framework for helping developers write tests. As the name tells SUnit was primarly developed for unit testing, but can also be used for integration tests. It is very similiar to other xUnit testing frameworks. The framework has 4 main classes: `TestCase`, `TestSuite`, `TestResult`, `TestResource`. `TestCase` is the main testing class. It's subclasses specify a set of test for a particuliar class or a system. Each test is represented by a method that starts with 'test' prefix. The tests are run by creating a new instance of the test subclass for every test method, running the setUp method, executing the test method and finally running the tearDown method for each of the test methods. [5] I did not need to use the `TestSuite`, `TestResult` and `TestResource` in my tests, as these classes are used when dealing with either limited resources (like database handles) or resources that take a long time to set up.

7.1.4 Implemented Tests

All the test classes implemented as of now include:

- `OPOCLParserTest`
- `OPOCLASTBuilderTest`
- `OPOCLEnvironmentTest`
- `OPOCLValueTest`
- `OPOCLBooleanValueTest`
- `OPOCLExpressionEvaluatorTest`

The naming follows a common convention of suffixing the class that is tested with the 'Test' suffix. The test classes are stored in the `OP-OCL-Interpreter` package under the Tests tag. We can categorize the test classes among unit and integration tests. Unit tests: `OPOCLEnvironmentTest`, `OPOCLValueTest`, `OPOCLBooleanValueTest`. The rest are integration tests. In figure 7.1 we can see the Pharo Test Runner which is a SUnit browser of all avaiable tests in the Pharo image. If we run the tests using the 'Run Coverage' button seen



OPOCLASTBuilderTest
OPOCLBooleanValueTest
OPOCLEnvironmentTest
OPOCLExpressionEvaluatorTest
OPOCLParserTest
OPOCLValueTest

38 run, 38 passes, 0 skipped, 0 expected failures, 0 failures, 0 errors, 0 unexpected passes

Figure 7.1: Pharo Test Runner running the implemented tests

in figure 7.1 and select the `OP-OCL-Interpreter` package as the packge to count the coverage against, we get a total coverage of 43%, which certainly is not great. If given more time it would be more than adequate to implement more tests. One upside is that at least the `OPOCLParserTest` class tests all the possible production rules in the handwritten grammar. I prioritized this test class, because I assumed that there are more skilled Pharo developers than there are SmaCC users and making changes to the grammar could be daunting for someone unexperienced with SmaCC.

7.2 OntoUML Use Case

The last objective remaining from the thesis goal is the OntoUML use case, where we show the ability of the interpreter to interpret an OntoUML constraint written in OCL and evaluate if the model does not violate the constraint. The OntoUML use case can be thought of as one large integration test, that is testing the entirity of the implementation, using its every part and subsystem.

7.2.1 OntoUML

OntoUML is an extension of UML used for conceptual modeling. OntoUML is based on Universal Foundational Ontology (UFO). It takes inspiration from different fields like: cognitive sciences, modal and mathematical logic, set theory and relations. Even though it is an extension of UML it actually removes lots of the UML defined terms. Its strength is in the fact that the aspects it adds to the UML allows it to be much more expressive and readable thus understandable. [9]

7.2.2 OntoUML Constraint in OCL

Without going into too many details, these are some of the class stereotypes that OntoUML defines: Kind, Subkind, Role, Phase, Category, RoleMixin, Mixin, Relator, Mode, Quality, Collective, Quantity. Each of these stereotypes have a special purpose in an OntoUML model and define their own sets of constraint that must hold true for an OntoUML model to be valid. [10]

I have chosen to test the OCL interpreter on upholding the Kind stereotype's three constraints. The OCL constraint now follows with description:

```
 1  context Model
 2  inv KindConstraints:
 3      let kindElements: Bag(Class) = packagedElements->select(
 4          each |
 5          each.appliedStereotypes->exists(
 6              stereotype |
 7              stereotype.oclIsKindOf(Kind)
 8          )
 9      ) in
10          kindElements->select(
11              each |
12              let parents: Bag(Class) = each.allParents in
13              ...
```

Listing 7.1: Context declaration for the OntoUML constraints

On line 1 of listing 7.1 we see the context to be declared on the Model itself, since this is a metamodel constraint. On lines 3 to 8 we define a 'kindElements' bag of UML classes that have at least one stereotype Kind applied. We then go through each of these elements one by one and for each define a 'parents' bag of classes that represents all the supertypes direct and indirect.

Let's now list the constraints definitions that we want to check[10] :

1. Kind cannot have an identity provider as their ancestor, that is: Kind, Collective, Quantity, Relator, Mode and Quality.
2. Kind cannot have types that inherit identity as their ancestor, that is: Subkind, Role, Phase.
3. Kind cannot have any anti-rigid type as their ancestor, that is: Role, RoleMixin, Phase.

```
        ...
        parents->exists( parent |
            parent.appliedStereotypes->exists(
                stereotype |
                stereotype.oclIsKindOf(Kind) or
                stereotype.oclIsKindOf(Collective) or
                stereotype.oclIsKindOf(Quantity) or
                stereotype.oclIsKindOf(Relator) or
                stereotype.oclIsKindOf(Mode) or
                stereotype.oclIsKindOf(Quality)
            )
        ) or
        parents->exists( parent |
            parent.appliedStereotypes->exists(
                stereotype |
                stereotype.oclIsKindOf(Subkind) or
                stereotype.oclIsKindOf(Role) or
                stereotype.oclIsKindOf(Phase)
            )
        ) or
        parents->exists( parent |
            parent.appliedStereotypes->exists(
                stereotype |
                stereotype.oclIsKindOf(RoleMixin) or
                stereotype.oclIsKindOf(Role) or
                stereotype.oclIsKindOf(Phase)
            )
        )
    ).size() = 0
```

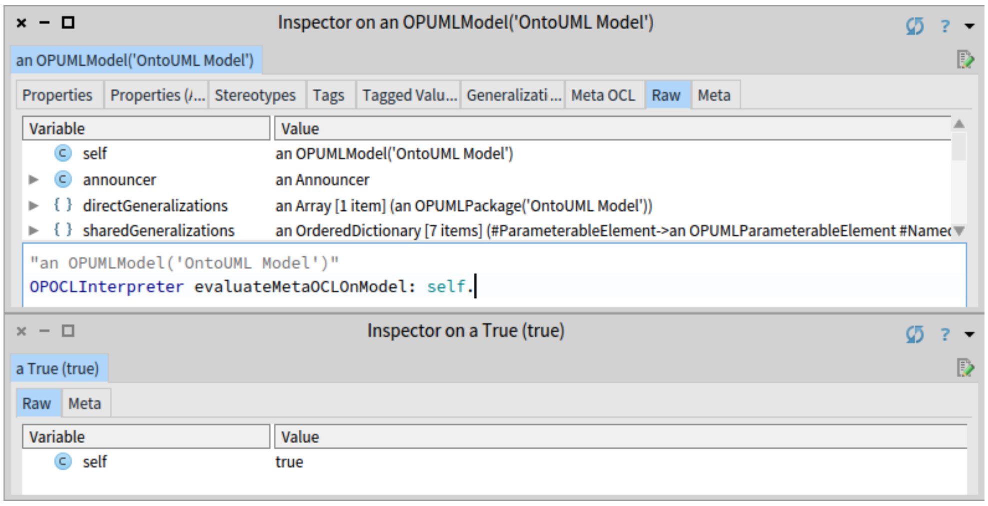
Listing 7.2: Searching for ancestors that break the constraint

In listing 7.2 every `parents->exists(...)` corresponds to one of the constraint. We are basically searching for any parent that has a stereotype that breaks the constraint and adding it into the collection. At the end we declare that such a collection should have its size equal to zero.

7.2.3 Evaluating the OCL constraint

To evaluate the constraint we first have to type it in the Meta OCL of some model. Let's open OpenPonk's OntoUML example model: `World Menu` → `OntoUML UFO-A Editor` → `Role Mixin`. Now in the 'Model Tree' on the bottom-left right-click the 'OntoUML Model' and select 'Inspect'. Go to the 'Meta OCL' tab and paste the OCL constraint there. Now go to the 'Raw' tab and in the bottom pane write and evaluate '`OPOCLInterpreter evaluateMetaOCLOnModel: self`'

In figure 7.2 we see what happens if we open up an inspector on the result of evaluating the OCL OntoUML constraint on a valid model. In figure 7.3 we modified the model so that the class Person which is of stereotype Kind


Inspector on an OPUMLModel('OntoUML Model')

an OPUMLModel('OntoUML Model')

Properties | Properties (/... | Stereotypes | Tags | Tagged Valu... | Generalizati... | Meta OCL | Raw | Meta

Variable	Value
self	an OPUMLModel('OntoUML Model')
announcer	an Announcer
directGeneralizations	an Array [1 item] (an OPUMLPackage('OntoUML Model'))
sharedGeneralizations	an OrderedDictionary [7 items] (#ParameterableElement->an OPUMLParameterableElement #Namec

```
"an OPUMLModel('OntoUML Model')"
OPOCLInterpreter evaluateMetaOCLOnModel: self.
```

Inspector on a True (true)

a True (true)

Raw | Meta

Variable	Value
self	true

Figure 7.2: Inspecting result of evaluating a valid model

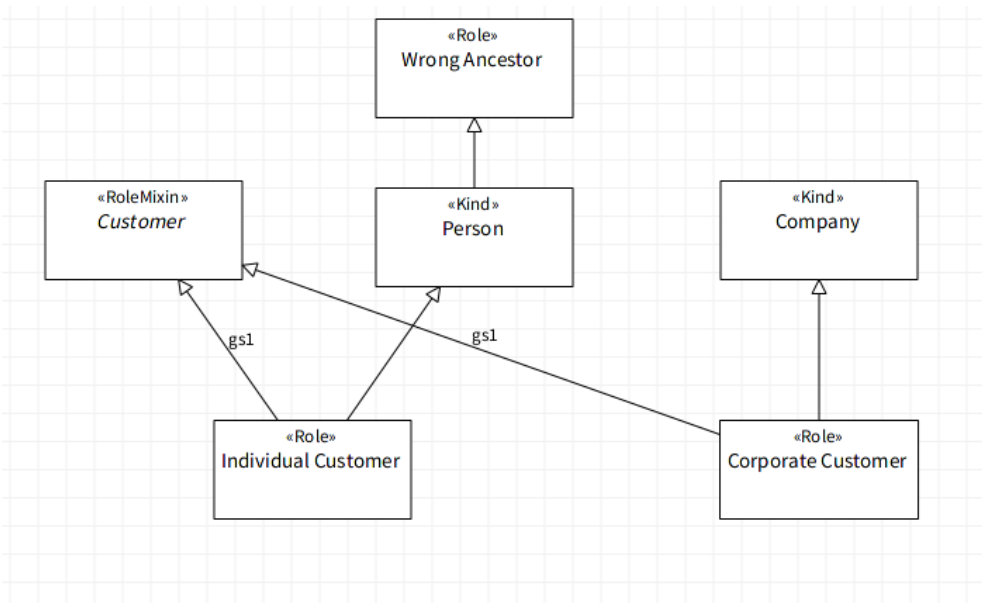
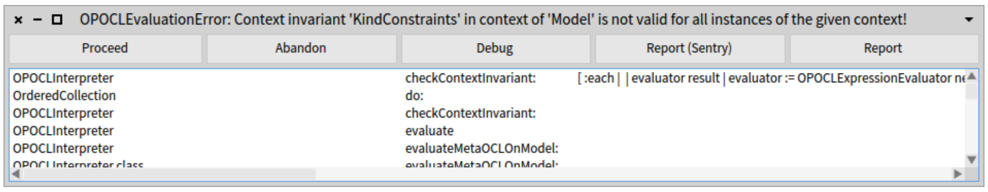


Figure 7.3: Modified model

inherits from the class Wrong Ancestor which is of stereotype Role clearly breaking the 2nd and 3rd Kind constraint.

In figure 7.4 we see what happens if we evaluate the OCL constraint on the modified model. An error pops up with the error message: "OPOCLEvaluatorError: Context invariant 'KindConstraints' in context of 'Model' is not valid for all instances of the given context!". This error does not actually tells



```
OPOCLEvaluationError: Context invariant 'KindConstraints' in context of 'Model' is not valid for all instances of the given context!
```

```
OPOCLInterpreter        checkContextInvariant:     [ :each | | evaluator result | evaluator := OPOCLExpressionEvaluator ne
OrderedCollection       do:
OPOCLInterpreter        checkContextInvariant:
OPOCLInterpreter        evaluate
OPOCLInterpreter        evaluateMetaOCLOnModel:
OPOCLInterpreter class  evaluateMetaOCLOnModel:
```

Figure 7.4: The error message when evaluating the constraint on the modified model

us where exactly the context invariant is invalidated, but that is because we defined the invariant on the model itself, so we just know that the invariant is invalid for the model. For future work, we should be able to define a Kind related invariant on the `Kind` stereotype itself. Then the error would print the concrete Kind, where the invariant is invalid.

We can see the OCL interpreter manages to evaluate whether the Kind constraints are upheld or not, and the demonstration was successfull.

Conclusion

The goal of this thesis was to create an OCL interpret capable of evaluating a subset of the OCL language, integrating it in the OpenPonk platform, testing it and showing that it is capable of ensuring an OntoUML model constraint.

First we looked at the OCL language as a whole, including the expressions, context declarations and the OCL standard library. We studied parsers and parsing algorithms, both traditional and alternative. We went over the basics of the Pharo programming language and the Pharo environment. We got familiar with Pharo's parser frameworks, PetitParser and more throughly with SmaCC. This concluded the research part of the thesis.

After research we went over some ideas, approaches and implementation attempts that did not work out. We also explained why these approaches failed. We then decided on the solution and what the OCL interpreter should support.

In the implementation part of the thesis we went over the packages, what the general purpose of the classes in them is and listed a few important ones. We also looked at the overview of the whole interpretation process.

With the implementation and tests done we successfully demonstrated the functionality of the OCL interpreter on ensuring the OntoUML model constraint.

The goal of the thesis was accomplished. The final implementation has some limitations and shortcomings, but those were met with suggestions and possible solutions, that could be implemented in the future. The OCL interpreter as it is in its current state is a great basis for further improvement and could be later used for improving the quality of models and modeling in general. Overall I am satisfied with the results.

Future Development

There are many ways to extend and add features to the OCL interpreter. Here is a list with some additional possible extensions:

- Right now the interpreter just evaluates OCL constraints that are defined on a model. The interpreter should definitely have the ability to evaluate OCL constraints that are defined on any model element.
- The handwritten grammar could be extended to handle more expression types, context and package declarations.
- Similiarly the AST could be extended, which would be used by the expression evaluator to handle the handwritten grammar extensions.
- Better integration in the OpenPonk platform itself, using Graphical User Inteface (GUI) elements
- Integrating the expression and type packages into UML.
- Better error messages with more information and better user feedback in general, including interactive errors that show what is wrong directly in the UML model being evaluated.
- Enhance the quality of integration and unit tests. Increase test coverage.

Bibliography

[1] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools.* 2nd ed. Greg Tobin, 2007. ISBN: 9780321486813.

[2] Anon. *Pharo The immersive programming experience.* 2018. URL: `http://pharo.org/web` (visited on 05/07/2018).

[3] Anon. *Smalltalk Syntax In A Postcard.* 2014. URL: `http://wiki.c2.com/?SmalltalkSyntaxInaPostcard` (visited on 05/07/2018).

[4] Alexandre Bergel et al. *Deep Into Pharo.* 1st ed. Lulu.com & Square Bracket Associates, 2013. ISBN: 9783952334164.

[5] Andrew P Black et al. *Pharo by example.* 1st ed. Lulu.com & Square Bracket Associates, 2010. ISBN: 9781365654596.

[6] John Brant. *SmaCC.* URL: `http://refactoryworkers.com/SmaCC.html` (visited on 05/08/2018).

[7] Jordi Cabot. *Object Constraint Language (OCL) tutorial.* 2012. URL: `https://modeling-languages.com/ocl-tutorial` (visited on 05/05/2018).

[8] Jordi Cabot. *Why you need to learn OCL.* 2012. URL: `https://modeling-languages.com/why-you-need-to-learn-ocl/` (visited on 05/07/2018).

[9] CCMi. *OntoUML.* 2016. URL: `https://ccmi.fit.cvut.cz/metodiky/ontouml/` (visited on 05/06/2018).

[10] OntoUML Community. *OntoUML Wiki - Kind.* 2017. URL: `https://ontouml.org/ufo/wiki/kind` (visited on 05/06/2018).

[11] Bryan Ford. “Packrat parsing:: simple, powerful, lazy, linear time, functional pearl”. In: *ACM SIGPLAN Notices.* Vol. 37. 9. ACM. 2002, pp. 36–47.

[12] Bryan Ford. “Parsing expression grammars: a recognition-based syntactic foundation”. In: *ACM SIGPLAN Notices*. Vol. 39. 1. ACM. 2004, pp. 111–122.

[13] Thierry Goubier et al. *SmaCC: a Compiler-Compiler*. `https://github.com/SquareBracketAssociates/Booklet-Smacc`. Booklet. 2017.

[14] Scott McPeak and George C Necula. “Elkhound: A fast, practical GLR parser generator”. In: *International Conference on Compiler Construction*. Springer. 2004, pp. 73–88.

[15] Bob Moore. “Removing Left Recursion from Context-Free Grammars”. In: Association for Computational Linguistics, 2000. URL: `https://www.microsoft.com/en-us/research/publication/removing-left-recursion-from-context-free-grammars/`.

[16] *Object Constraint Language*. formal/2014-02-03. Rev. 2.4. Object Managment Group. Feb. 2014.

[17] *OMG Unified Modeling Language (OMG UML)*. formal/2017-12-05. Rev. 2.5.1. Object Managment Group. Dec. 2017.

[18] Roy Osherove. *The Art of Unit Testing*. 2nd ed. Manning Publications Co., 2014. ISBN: 9781617290893.

[19] Martyn A Ould and Charles Unwin. *Testing in software development*. Cambridge University Press, 1986. ISBN: 9780521337861.

[20] S Doaitse Swierstra. “Combinator parsing: A short tutorial”. In: *Language Engineering and Rigorous Software Development*. Springer, 2009, pp. 252–300.

[21] Gabriele Tomassetti. *A Guide to Parsing: Algorithms and Technology (Part 1)*. URL: `https://dzone.com/articles/a-guide-to-parsing-algorithms-and-technology-part` (visited on 05/12/2018).

[22] Eelco Visser et al. *Scannerless generalized-LR parsing*. Universiteit van Amsterdam. Programming Research Group, 1997.

[23] Alessandro Warth, James R Douglass, and Todd D Millstein. “Packrat parsers can support left recursion”. In: *PEPM* 8 (2008), pp. 103–110.

Appendix **A**

Handwritten OCL Grammar

```
%glr;
%unicode;

# Creates a starting state for each of these non-terminals, important for tests
%start ContextInvariant OCLExpression VariableDeclaration
    VariableDeclarationList SimpleName Arguments Type IterationName;

# SmaCC cannot deal with unicode greater than \xFFFF
<NameStartChar>
    : [A-Z] | _ | $ | [a-z]
    | [\xC0-\xD6] | [\xD8-\xF6] | [\xF8-\x2FF]
    | [\x370-\x37D] | [\x37F-\x1FFF]
    | [\x200C-\x200D] | [\x2070-\x218F] | [\x2C00-\x2FEF]
    | [\x3001-\xD7FF] | [\xF900-\xFDCF] | [\xFDF0-\xFFFD]
    ;
<NameChar>: (<NameStartChar> | [0-9]);
<Identifier>: <NameStartChar> <NameChar>*;
<Digit>: [0-9];
<LeadingDigit>: [1-9];
<IntegerLexicalRepresentation>: (0 | <LeadingDigit> <Digit> *);
<IntegerPart>: <IntegerLexicalRepresentation> ;
<FractionPart>: \. <Digit> + ;
<ExponentPart>: [eE] [\+\-]? <IntegerPart> ;
<RealLexicalRepresentation>: <IntegerPart> (
    (<FractionPart> <ExponentPart>?) | (<FractionPart>? <ExponentPart>)
);
<Char>: [\x20-\x26] | [\x28-\x5B] | [\x5D-\xD7FF] | [\xE000-\xFFFD];
<EscapeSequence>: \\ (([btnfr\"\'\\]) | (x <Digit> <Digit>)
    | (u <Digit> <Digit> <Digit> <Digit>));
<QuotedText>: \' (<Char> | <EscapeSequence>) * \';

# SmaCC ignores whitespaces by default
<whitespace>: \s+;
<comment>: (-- [^\n] * ) | (/\* [^"*/"]* \*/);

# These are defined, so we can set different
# %left/%right directive to Binary/Unary
```

```
<BinaryPlus>: \+;
<BinaryMinus>: -;
<UnaryPlus>: \+;
<UnaryMinus>: -;


# Operator precedence and some shift/reduce conflicts solving directives
%left "implies";
%left "xor";
%left "or";
%left "and";
%left "=" "<>";
%left "<" ">" "<=" ">=";
%left <BinaryPlus> <BinaryMinus>;
%left "*" "/";
%right "not" <UnaryMinus> <UnaryPlus> "_" ;
%left "." "->" "^" "^^";
%right <Identifier> <QuotedText>;

# Setting the prefixes and suffixes of the generated
# CST class hierarchy
%root CST;
%prefix OPOCL;
%suffix Node;

# Makes unnamed tokens in production rules to be added
# to the generated class, useful for debugging/error info
%annotate_tokens;

# Defining the CST hierarchy
%hierarchy OCLExpression (Navigation Call Literal Operator Let);
%hierarchy Navigation (ImplicitNavigation);
%hierarchy Call (ImplicitCall);
%hierarchy Literal (Integer UnlimitedNatural String Real Boolean);
%hierarchy Operator (BinaryOperator UnaryOperator);
%hierarchy Type (RegularType CollectionType);

# Every CST class instance has to have an AST counter-part
%attributes CST (ast);

ContextInvariant
   : "context" SimpleName 'contextName' "inv" (SimpleName 'invariantName')?
     ":" OCLExpression 'invariantConstraint' {{}}
   ;

OCLExpression
   : Navigation
   | Call
   | Iteration
   | Literal
   | BinaryOperator
   | UnaryOperator
   | Parantheses
   | Let
```

```
    | IfThenElse
    ;

Navigation
    : ImplicitNavigation
    | OCLExpression 'origin' "." SimpleName 'identifierName' {{}}
    ;

ImplicitNavigation
    : SimpleName 'identifierName' {{}}
    ;

Call
    : ImplicitCall
    | OCLExpression 'origin' "." SimpleName 'operationName'
      "(" Arguments 'arguments'  ?")" {{}}
    | OCLExpression 'origin' "->" SimpleName 'operationName'
      "(" Arguments 'arguments'  ?")" {{}}
    ;

ImplicitCall
    : SimpleName 'operationName' "(" Arguments 'arguments' ? ")" {{}}
    ;

Iteration
    : OCLExpression 'origin' "->" IterationName 'iterationName'
      "("  (VariableDeclaration 'iterator1'
      ("," VariableDeclaration 'iterator2' )? "|" )?
      OCLExpression 'bodyExp' ")" {{}}
    ;

IterationName
    : "select" 'nameToken' {{}}
    | "exists" 'nameToken' {{}}
    ;

SimpleName
    : <Identifier> 'nameToken' {{}}
    | "_" <QuotedText> 'nameToken' {{}}
    | SimpleName <QuotedText> 'nameToken' {{}}
    ;

Literal
    : Integer
    | UnlimitedNatural
    | String
    | Real
    | Boolean
    ;

Integer
    : <IntegerLexicalRepresentation> 'integerValueToken' {{}}
    ;
```

```
UnlimitedNatural
    : ”∗” ’unlimitedNaturalValueToken’ {{}}
    ;

String
    : <QuotedText> ’textToken’ {{}}
    | String  <QuotedText> ’textToken’ {{}}
    ;

Real
    : <RealLexicalRepresentation> ’realValueToken’ {{}}
    ;

Boolean
    : ”true” ’booleanValueToken’ {{}}
    | ”false” ’booleanValueToken’ {{}}
    ;

BinaryOperator
    : OCLExpression ’leftExp’ <BinaryPlus> ’binaryOperatorToken’
      OCLExpression ’rightExp’ {{}}
    | OCLExpression ’leftExp’ <BinaryMinus> ’binaryOperatorToken’
      OCLExpression ’rightExp’ {{}}
    | OCLExpression ’leftExp’ ”∗” ’binaryOperatorToken’
      OCLExpression ’rightExp’ {{}}
    | OCLExpression ’leftExp’ ”/” ’binaryOperatorToken’
      OCLExpression ’rightExp’ {{}}
    | OCLExpression ’leftExp’ ”implies” ’binaryOperatorToken’
      OCLExpression ’rightExp’ {{}}
    | OCLExpression ’leftExp’ ”xor” ’binaryOperatorToken’
      OCLExpression ’rightExp’ {{}}
    | OCLExpression ’leftExp’ ”and” ’binaryOperatorToken’
      OCLExpression ’rightExp’ {{}}
    | OCLExpression ’leftExp’ ”or” ’binaryOperatorToken’
      OCLExpression ’rightExp’ {{}}
    | OCLExpression ’leftExp’ ”=” ’binaryOperatorToken’
      OCLExpression ’rightExp’ {{}}
    | OCLExpression ’leftExp’ ”<>” ’binaryOperatorToken’
      OCLExpression ’rightExp’ {{}}
    | OCLExpression ’leftExp’ ”<” ’binaryOperatorToken’
      OCLExpression ’rightExp’ {{}}
    | OCLExpression ’leftExp’ ”>” ’binaryOperatorToken’
      OCLExpression ’rightExp’ {{}}
    | OCLExpression ’leftExp’ ”<=” ’binaryOperatorToken’
      OCLExpression ’rightExp’ {{}}
    | OCLExpression ’leftExp’ ”>=” ’binaryOperatorToken’
      OCLExpression ’rightExp’ {{}}
    ;

UnaryOperator
    : ”not” ’unaryOperatorToken’ OCLExpression ’exp’ {{}}
    | <UnaryPlus> ’unaryOperatorToken’ OCLExpression ’exp’ {{}}
    | <UnaryMinus> ’unaryOperatorToken’ OCLExpression ’exp’ {{}}
    ;
```

```
Parantheses
    : ”(” OCLExpression ’exp’ ”)” { ˆ exp }
    ;

Let
    : ”let” VariableDeclarationList ’variableDeclarations’
      ”in” OCLExpression ’exp’ {{}}
    ;

IfThenElse
    : ”if” OCLExpression ’condition’ ”then” OCLExpression
      ’thenExp’ ”else” OCLExpression ’elseExp’ ”endif” {{}}
    ;

VariableDeclarationList
    : VariableDeclaration ’declaration’ {{}}
    | VariableDeclarationList ”,” VariableDeclaration ’declaration’ {{}}
    ;

VariableDeclaration
    : SimpleName ’variableName’  (”:” Type ’variableType’)?
      (”=” OCLExpression ’initExp’)? {{}}
    ;

Type
    : RegularType
    | CollectionType
    ;

RegularType
    : SimpleName ’typeName’  {{}}
    ;

CollectionType
    :  SimpleName ’collectionTypeName’ ”(” Type ’elementType’ ”)”  {{}}
    ;

Arguments
    : OCLExpression ’argument’ {{}}
    | Arguments ”,” OCLExpression ’argument’ {{}}
    ;
```
Listing A.1: The entire handwritten grammar for the OCL language subset

Appendix B

Acronyms

AST	Abstract Syntax Tree.
CCMi	Center for Conceptual Modeling and Implementation.
CFG	context-free grammar.
CST	Concrete Syntax Tree.
GLR	Generalized LR.
GUI	Graphical User Inteface.
LALR	Look-Ahead LR.
LL	Left to right, Leftmost derivation.
LR	Left to right, Rightmost derivation.
MDE	model-driven engineering.
OCL	Object Constraint Language.
OMG	Object Managment Group.
PEG	parsing expression grammar.
SmaCC	Smalltalk Compiler-Compiler.
UFO	Universal Foundational Ontology.
UML	Unified Modeling Language.
VM	virtual machine.

Appendix C

User Guide

This guide details how to run Pharo with the OpenPonk image containing the OCL interpreter. For reference of the directory structure see at appendix D.

C.1 Linux

Copy the `/exe/lin` directory from the USB flash disk to a location on your computer, where you have write privilages.

`cd` into the `lin` directory on your computer and run the command:

```
./vms/linux/pharo openponk.image
```

The version of the Pharo VM is 32-bit, as it is the most stable one. You may have to install the 32-bit runtime libraries. Run the following commands to install those:

```
sudo dpkg --add-architecture i386 sudo apt-get update
sudo apt-get install libx11-6:i386 libgl1-mesa-glx:i386 \
libfontconfig1:i386 libssl1.0.0:i386
sudo apt-get install libcairo2:i386
```

OpenPonk also requires libcairo to be installed (the 32-bit version). Run the following command:

```
sudo apt-get install libcairo2:i386
```

C.2 Windows

Copy the `/exe/win` directory from the USB flash disk to a location on your computer, where you have write privilages.

In the File Explorer go to your copied `win` directory and double-click the `Pharo.exe` file. This should automatically start Pharo with an image running. If it prompts for an image, just select the `openponk.image` file in the same directory.

C.3 Mac

For running Pharo on a Mac you can follow the guide on the `http://pharo.org/download` web page.

Then you can use the `openponk.image` file, that is stored in the `/exe/img/` directory stored in the USB flash disk.

Appendix **D**

Contents of enclosed USB

`readme.txt`	description of the USB contents and a user guide
└─ `exe`	directory with Linux VM and image
│ └─ `lin`	directory with Windows VM and image
│ └─ `img`	directory with the OpenPonk OCL Interpreter image
│ └─ `win`	directory with executables
└─ `src`	directory of source codes
│ └─ `ocl_interpreter`	directory with Pharo file-outs
│ │ └─ `OP-OCL-Interpreter.st`	OCL Interpreter package file-outs
│ │ └─ `OPUMLMetaElement.st`	OpenPonk modification file-out
│ └─ `thesis`	directory of LaTeX source codes of the thesis
└─ `text`	thesis text directory
│ └─ `BP_Svoboda_Jakub_2018.pdf`	thesis text in PDF format
│ └─ `BP_Svoboda_Jakub_2018.ps`	thesis text in PS format