# The Overture Project

Designing an open source tool set

BY
Pieter van der Spek

August 2004


Electrical Engineering, Mathematics and Computer Science
Information, Systems and Algorithms
Software Engineering
http://www.ewi.tudelft.nl


West Consulting BV
Delftechpark 5
2628 XJ Delft
+31 (0)15 219 1600
http://www.west.nl

# The Overture Project

Designing an open source tool set

MASTER'S THESIS

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER SCIENCE

Work carried out at
West Consulting BV, Delft

BY

Pieter van der Spek

August 2004

# Graduation data

---

| | | |
|---|---|---|
| Author | : | Pieter van der Spek |
| Title | : | The Overture Project; Designing an open source tool set |
| Date | : | August 10, 2004 |
| Location | : | Faculty EWI, Mekelweg 4, Delft |

**Graduation committee:**

| | |
|---|---|
| Prof. dr. A. van Deursen | TU Delft (chairman) |
| Ir. C. Pronk | TU Delft |
| Dr. K. G. Langendoen | TU Delft |
| Dr. ir. N. Plat | West |

**Abstract:**

Overture is a formal modeling method that is currently being developed. This thesis describes the process of the design and implementation of a development environment for the Overture Modeling Language which is part of Overture.
The set of tools that is the result of this effort consists of three components. The first and most visible component is a front-end, integrated in the Eclipse Platform. This front-end consists of an editor and some functionality to supply information to the user (e.g. error messages).
The second part is a compiler. This part handles the verification of the syntax and the semantics of the user input.
The third and final part consists of an XML processor. This part transforms the original user input into an XML representation. Also, the processor is able to validate the XML representation with the help of an XML Schema.

# Preface

This Master's thesis is a part of the Master of Science (MSc) degree in Computer Science of the Technical University of Delft, Electrical Engineering, Mathematics and Computer Science faculty, department of Information Systems Algorithms, Software Engineering group.

The project was carried out at West Consulting (WEST) which is a leading high-end IT development, integration and consulting company.

I would like to take this opportunity to give a special thanks to my advisers Kees Pronk at the Technical University of Delft, and Nico Plat at WEST for their continuous support and critical readings of this paper.
I also want to thank the staff of WEST for their support and kindness. In particular, I want to thank Edward Akerboom, Arthur de Jong and Gertjan van Oosten for reviewing this thesis.
Finally, I want to thank John Fitzgerald, Paul Mukherjee, Peter Gorm Larsen and Marcel Verhoef for their support and advice.

Delft, August 2004,
Pieter van der Spek

# Contents

# List of Figures

# List of Tables

# Listings

# Chapter 1

# Introduction

The limitations of graphical models in object-oriented analysis are widely accepted. While superior for visualisation, a graphical model is not enough for precise and unambiguous specification. Precisely for this reason formal modeling techniques have been developed. One such modeling method is known as VDM++. Although the strength and usefulness of this technique is widely acknowledged, in practice it is not being used extensively.
In order to help raise the popularity of formal modeling techniques in industrial software engineering projects, a group of people (namely John Fitzgerald, Paul Mukherjee, Peter Gorm Larsen, Nico Plat and Marcel Verhoef) have taken up the task to modify VDM++. They aim to create a modeling technique that has the potential to get the industry to use and familiarise itself with these techniques and thus raise the standard of their software engineering practices.
This has led to the initiative of the Overture project. The objective of the Overture project is to provide an industrial-strength, open source tool set to allow the use of precise abstract models (expressed in the Overture Modeling Language) in software development, and to foster an environment that allows researchers and other interested parties to experiment with modifications and extensions to the tool. This tool set will be called the "Overture Development Tools" or ODT for short (the rationale behind this name will be explained in chapter 2).

## 1.1 Aim of this project

In a research study that was recently carried out as part of this project [Spe04], an overview is provided of the tools and techniques that can be used in order to create the tool set that has been mentioned above.
The aim of the part of the project, as described in this document, is to design and implement part of the tool set. The components that have been created are the following:

1. An editing environment;
2. A syntax checker and a (partial) implementation of a static semantics checker;
3. An XML processor, which is able to generate XML from the original input, as provided by the user. Also functionality is available to validate the XML instance documents against an XML Schema.

## 1.2 Outline of this document

Chapter 2 gives a global overview of the Overture Development Tools.
Subsequently, chapter 3 will deal with the syntactic structure of the Overture Modeling Language (OML).
Following the discussion on the syntactic structure of OML, chapter 4 discusses some implementation issues that arose during the construction of the syntax and static semantics checker. Much of the discussion is focused on writing a compiler using a Java-based compiler generator.
Chapter 5 is devoted to creating an XML schema for the Overture Modeling Language. Also, the actual creation of XML instance of documents receives some attention.
Chapter 6 concludes and give an overall view of the achievements of this project.
Finally, in chapter 7, some thought is given to the remaining tasks necessary make this tool set a fully functional set of the Overture Development Tools. Also, some possible extensions to the core tools are mentioned.

## 1.3 Prerequisites

The reader is strongly advised to read [Spe04]. This document contains introductions to most of the subjects that are treated in this document. Also, the rationale behind the use of the various tools that are necessary for this project have been explained there.
Next to that, general knowledge of computer science, programming languages, design and construction of compilers for programming languages and knowledge of XML and XML related technologies will make reading this document easier.

# Chapter 2

# Overture Development Tools

One of the goals of the Overture project is to provide an environment which allows the use of precise abstract models (expressed in the Overture Modeling Language). This goal can be realized by extending an existing tool platform, called Eclipse (`http://www.eclipse.org`), with a set of tools (or plugins) that enable a user to create specifications using the Overture Modeling Language. In analogy to the names of existing toolkits in Eclipse, namely the Java Development Tools (JDT) and C/C++ Development Tools (CDT), the set of tools for the Overture Modeling Language will be called the Overture Development Tools (ODT).
Before diving into the deep end of the Overture Development Tools, this chapter will discuss the overall design of the tool set and the points at which the separate parts, which are discussed in the next chapters, come together.

## 2.1 Design of the ODT

The guiding principle in the architecture of the tool is the plugin structure of Eclipse. The Overture Development Tools will be built as a collection of plugins. Each of the parts will be created as a separate plugin to Eclipse. Other developers can create their own plugins to perform some specific task related to an Overture specification.
The general idea behind the ODT plugin structure can be seen in figure 2.1. This figure shows the three main plugins of the ODT (i.e. the editor, the

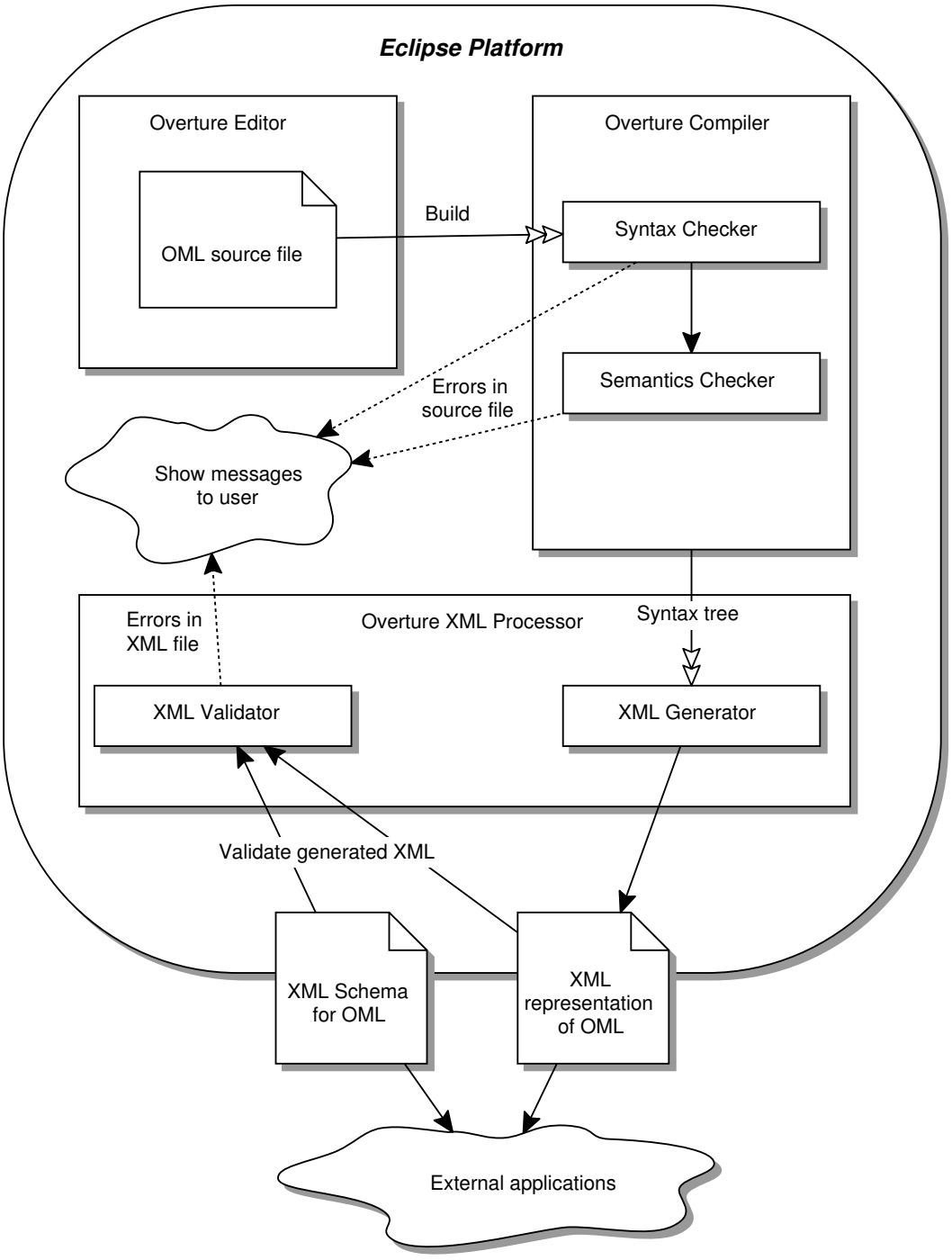


Figure 2.1: Flow of information through the ODT.

compiler and the XML processor). It also shows how information flows between the plugins. The two white, doubleheaded arrows indicate information flows that can be initiated by the user.

### 2.1.1 Communication between plugins

An important aspect in the design is the way in which different tools communicate with each other. There are several alternatives. The first option is, of course, using the input given by the user directly. Although this is very easy to realize, it is probably not the most efficient way. The input is likely to be error prone, which is an undesirable property of a means of communication. More suitable candidates are the syntax tree (either concrete or abstract) that is generated by the compiler and the XML representation of the original input, both of which are shown in figure 2.1. One of the advantages of the use of the syntax tree is that it is always available within the Eclipse environment. As soon as the user creates a new class, the compiler generates the corresponding tree structure. The tree structure can be made available to developers through interfaces that define tree walking functionality.
On the other hand, the syntax tree can be too large or too inefficient for certain specific tasks. Especially when it is necessary to walk the entire structure (possibly even more than once) without the need of every little detail provided by the parse tree. In such an event, the XML representation might be an interesting alternative. It contains most of the information that is available in the parse tree (but not all of it). Next to that, standard implementations for reading an XML document exist that will allow a developer simple and fast access to the parts of the document of interest. Furthermore, tools working outside Eclipse do not have access to the syntax tree and have no other option than to use the XML document, This, by the way, is the reason why the Schema document and the instance document are placed on the edge of the Eclipse Platform in figure 2.1.
Therefore it seems no more than logical to allow both "communication channels" to be used, and let the specific circumstances decide which is the best option.

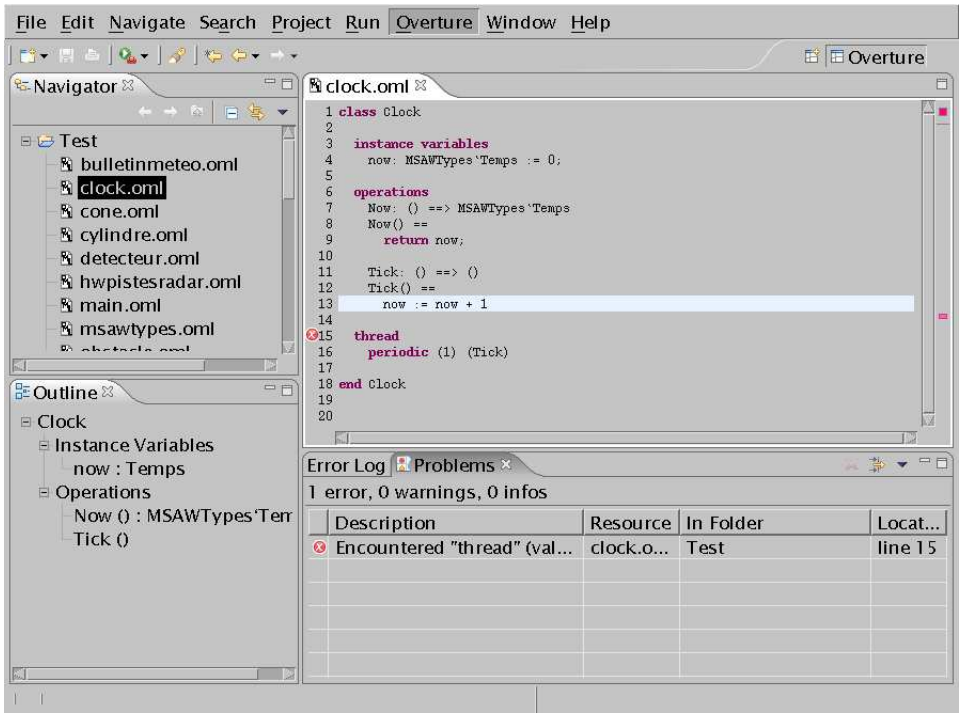


Figure 2.2: Overture environment in Eclipse.

## 2.2 The Overture Development Tools

Before going behind the scenes to see how the ODT is put together, it helps to have a sense of what the ODT does and what it looks like to the user. Figure 2.2 shows what the workbench normally looks like when the user is writing an Overture specification.
The Overture Development Tools add the capabilities of a full-featured Overture IDE to the Eclipse Platform (some of which are visible in figure 2.2). This means that it allows a user to create and modify specifications written using the Overture Modeling Language.
Additionally, features that are discussed futher on in this document, like the compiler functionality and the XML generator, have been made available inside the Eclipse Platform.

### 2.2.1 Implementation

The ODT is implemented as a group of plugins, with the user interface (UI), the compiler and the XML processor in separate plugins. This separation of UI, compiler and XML processor code allows the ODT compiler to be used by other tools that incorporate Overture capabilities but do not need the ODT UI.
The main features of the UI will be shown next. The compiler is described in chapter 4, whereas the XML processor is discussed in chapter 5.

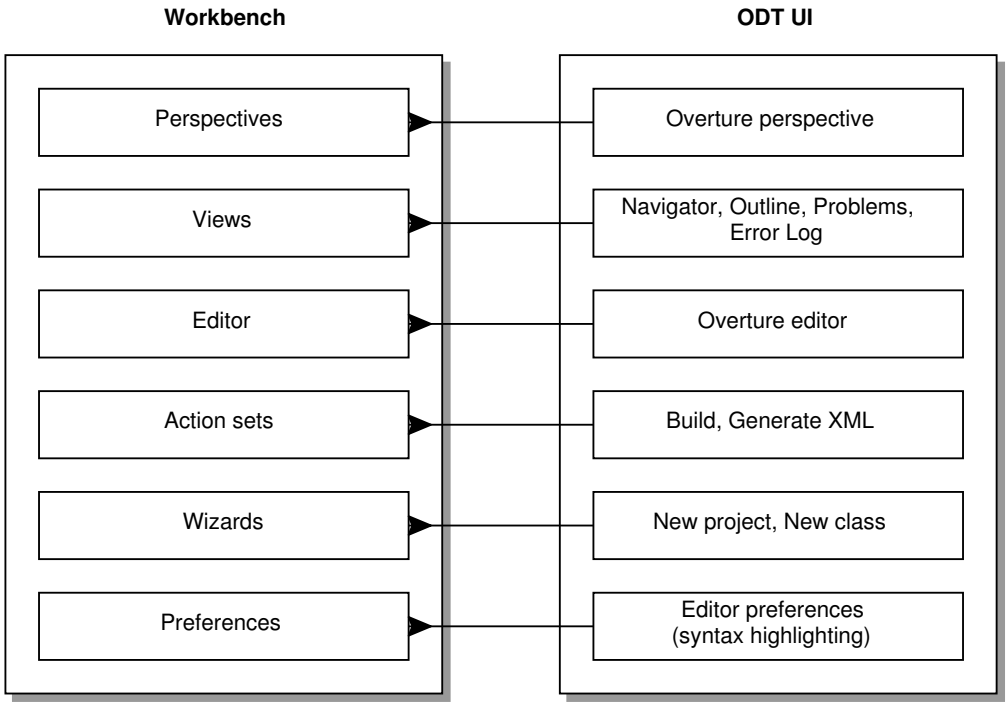


Figure 2.3: Key connections between ODT and Eclipse Platform.

Figure 2.3 illustrates key connections between the Eclipse Platform and the ODT UI. These elements will be discussed below.

**Perspective and views** The Overture perspective defines the default layout of the editor area and the surrounding views. It allows the user to easily access all files related to a specific process. Also, two views (i.e the Problems view and the Error Log) are available to inform the user of problems with the specification in the active editor.

**Editor** The editor enables the user to enter a specification in the Overture Modeling Language. The current implementation of this editor supports syntax highlighting of the keywords that are available in OML. Also, markers are placed on opposite sides of the editor area when a syntactic or semantic error is encountered, as can be seen in figure 2.2.

**Action sets** In order to enable the user to manually start the process of verifying the correctness of the contents of the active editor, a special menu item (name "Build") is available to do just that.
The second menu item that is available starts the process of generating XML from the contents of the active editor.

**Wizards and preferences** Dedicated wizards are available within Eclipse to enable users to create new Overture projects and Overture class files. The current implementation resembles very much the process of creating ordinary text files and standard Eclipse projects. Future implementations could support more advanced options (e.g. setting special features for a project).
Additionally, preference pages exist that allow a user to change the colour of certain parts (e.g. keywords or background) of the editor.

# Chapter 3

# Overture

Every programming language has rules that prescribe the syntactic structure of well-formed programs written in that language. This chapter will describe the syntactic structure of the Overture Modeling Language and some of the rules that define the well-formedness. Also, this chapter will give an introduction to subjects related to the syntax and semantics of a computer language, which will be used in subsequent chapters.

## 3.1 Context-free grammar

The syntax of programming language constructs can be described by context-free grammars (CFG) or Backus-Naur Form (BNF) notation.
A grammar can naturally describe the hierarchical structure of many programming languages. For example, an if-then-else-expression in OML has the form

if *expression* then *expression* else *expression*

That is, the if-then-else expression is the concatenation of the keyword `if`, an expression, the keyword `then` another expression, the keyword `else` and a third expression. Using the variable *expr* to denote an expression, this structuring rule can be expressed as

$$\mathsf{expr} \rightarrow \mathsf{if}\ expr\ \mathsf{then}\ expr\ \mathsf{else}\ expr$$

in which the arrow may be read as "can have the form". Such a rule is called a *production*. In a production elements like the keyword `if` are called *tokens*. Variables like `expr` represent sequences of tokens and are called *nonterminals*. More formally, a context-free grammar consists of four components [Aho86]:

1. A set of nonterminals (denoted by N);
2. A set of tokens, known as terminal symbols (denoted by T); these symbols are informally know as *reserved words* or *keywords* of a grammar;
3. A set of productions where each production consists of a nonterminal, called the left hand side of the production, and a sequence of tokens and/or nonterminals, called the right hand side of the production (denoted by P);
4. A designation of one of the nonterminals as the start symbol (denoted by S).

In addtion to these four components, there are three additional conditions that a grammar has to fulfill [Gru00]:

1. $N \cap T = \oslash$ : this condition says that the set of terminals and the set of nonterminals can not have any symbols in common (i.e. it must be possible to separate terminals from nonterminals);
2. $S \in N$: this conditions says that the start symbol must be a nonterminal;
3. $P \in \{ (M, \alpha) \mid M \in N, \alpha \in (N \cup T)^* \}$: this means that the left-hand side of each production (denoted by M) must be a nonterminal and that the right-hand side (denoted by $\alpha$) may consists of zero or more (denoted by *) terminals and non-terminals but is not allowed to include any other symbols.

Thus, the syntax of a programming language can be described by means of a grammar. However, grammars can be described at several levels of abstraction. The lowest level of abstraction is called the concrete syntax. The concrete syntax contains every terminal and nonterminal symbol of the

programming language. Although this level of detail can sometimes be useful (for example when you do not want to generate executable code but a nicely formatted version of the input), often the terminal symbols do not convey any extra information. A very simple example is the following type of expression:

$$\text{expr} \rightarrow \text{'"'} \; expr \; \text{'"'}$$

Obviously, the double quotes supply information to a human reader, but for analyzing the syntax of a programming language they aren't of much practical use. Also, less obvious examples exist. Take for example a VDM++ composite type:

$$\text{composite type} \rightarrow \text{'compose'} \; identifier \; \text{'of'} \; fieldlist \; \text{'end'}$$

Observe that this type is fully defined by the identifier and the field list. Therefore, it is possible to use an abstract syntax notation such as:

$$\text{mk\_CompositeType (identifier , field list )}$$

This expression (written as a VDM++ record type constructor) creates a record consisting of an identifier and a field list, which allows access to all important parts of this definition, without the superfluous information of the original definition ("syntactic sugar").

This observation is important because this, more abstract, syntax is used for the semantic analysis process. During semantic analysis certain checks are performed to ensure that the components of a program fit together meaningfully. So there is no need to keep track of the terminal symbols unless they are of specific interest for the semantics. Hence, for semantic analysis the high level of abstraction of the abstract syntax contains more than enough information.

## 3.2 Semantics

A programming language can be defined by describing what its programs look like (the syntax of a language) and what its programs mean (the semantics of a language). As we have seen, the syntax of a language can be concisely described using a context-free grammar or BNF. Given the notations currently available, the semantics of a language is much more difficult to describe than the syntax. Consequently, for specifying the semantics of a language informal descriptions and examples are used.
The term 'semantics' actually has two meanings. In our case, semantics refers to the static semantics of a programming language. The task of static semantic analysis is to determine those properties of programs that can be computed using only the program text [Wil95]. This kind of analysis can help to ensure that certain kinds of programming errors will be detected and reported when the input is being compiled. This is opposed to dynamic semantics, which refers to properties of programs which can only be determined when the compiled program is run [Wil95]. Thus checking dynamic sematics might help to detect and report errors during the execution of a target program.
Static semantic checks are carried out according to the semantic conventions of the programming language and thus verify if the developer created a well-formed program. Examples of static checks include [Aho86]:

1. *Type checks.* A compiler should report an error if an operator is applied to an incompatible operand; for example, if an array variable and a function variable are added together.

2. *Flow-of-control checks.* Statements that cause the flow of control to leave a construct must have some place to transfer the flow of control to. For example, a break statement in C causes control to leave the smallest enclosing while, for, or switch statement; an error occurs if such an enclosing statement does not exist.

3. *Uniqueness checks.* There are situations in which an object must be defined exactly once. For example, in Pascal, an identifier must be declared uniquely inside a scope, labels in a case statement must be distinct.

4. *Name-related checks.* Sometimes, the same name must appear two or more times. For example, in Ada, a loop or block must have a

name that appears at the beginning and the end of the construct. The compiler must check that the same name is used in both places.

Of these four, type checking is often the hardest to perform. As can be seen from the examples, many of the other checks can be incorporated into other parts of the compiler. Uniqueness checks, for example, can be carried out when the symbol table[1] is generated.
The design of a type checker for a language is based on information about the syntactic constructs in the language, the notion of types, and the rules for assigning types to a language construct. This information is commonly called the type system for that language [Aho86].

In the next paragraph we will take a look at the syntactic structure of the Overture Modeling Language and some of the related semantic rules..

## 3.3 The syntax and semantics of OML

The syntax of Overture Modeling Language (OML) constructs is defined using grammar rules. The notation that will be used throughout the next paragraph is the ASCII (also called interchange) concrete syntax, as opposed to the mathematical-like concrete syntax. Also, wherever the syntax for parts of the language is presented, it will be described in a BNF dialect. The BNF notation used employs the following special symbols:

[1] A symbol table is a data structure containing a record for each identifier in the input, with fields for the attributes of the identifier. These attributes may provide information about the storage allocated for an identifier, its type, its scope (where in the program it is valid), and, in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument (e.g. by reference), and the type returned, if any [Aho86].

| | |
|---|---|
| , | concatenate symbol |
| = | the define symbol |
| \| | the definition separator symbol (alternatives) |
| [ ] | enclose optional syntactic items |
| { } | enclose syntactic items which may occur zero or more times |
| ' ' | single quotes are used to enclose terminal symbols |
| meta identifier | non-terminal symbols are written in lower-case letters (possibly including spaces) |
| ; | terminator symbol to denote the end of a rule |
| ( ) | used for grouping, e.g. "a, (b \| c)" is equivalent to "a, b \| a, c" |
| - | denotes subtraction from a set of terminal symbols (e.g. "character - ('A')" denotes all characters except the character A) |

### 3.3.1 Structure of the language

Figure 3.1 gives a global overview of the grammar for the Overture Modeling Language. Arrows in this figure denote a dependency between the two "blocks" (in a sense that the block, from which the arrow originates, contains one or more blocks from the kind at which the arrow ends). Each arrow points to a block (represented by an oval in figure 3.1) which is contained within the block from which the arrow originates. The arrow that points from Statement to Expression is special in a sense that it is optional.
Although the high level of abstraction causes this overview to be not entirely correct, it does show some important aspects of the language. First, a document containing an Overture model is divided up in one or more classes. Secondly, a class itself also contains several blocks, which eventually end in an expression.
In the following sections each of the separate parts of the grammar is discussed[2]. The entire grammar is not given here. It can be found in appendix A.

[2] The rules given in this paragraph will not always be the rules as they have actually been used. See also par. 4.3.

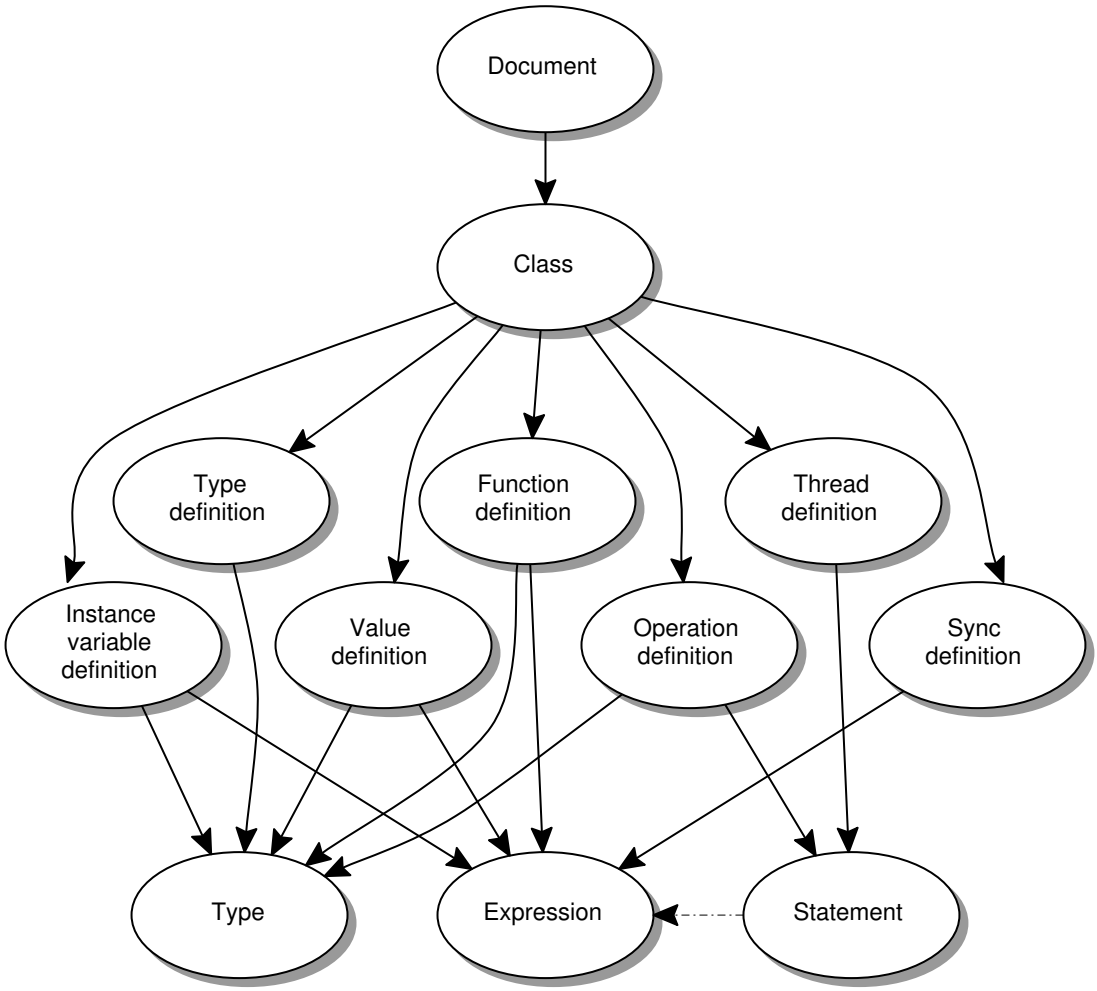


Figure 3.1: Global structure of OML grammar.

### 3.3.2 Documents and classes

The root (or starting symbol) of the grammar is the non-terminal 'document'. Directly located under this non-terminal in the hierarchy is the definition of a class. The primary rules of this part of the grammar are:

Listing 3.1: Documents and classes

```
1 document = class , { class } ;
2 class = 'class' , identifier ,
3       [ inheritance clause ] ,
```

```
4       [ class body ] ,
5       'end' , identifier ;
6   inheritance clause = 'is subclass of' ,
7       identifier , { ',' , identifier } ;
8   class body = definition block ,
9       { definition block } ;
10  definition block = instance variable definitions
11                   | type definitions
12                   | value definitions
13                   | function definitions
14                   | operation definitions
15                   | thread definitions
16                   | synchronization definitions ;
```

The class name, as given in the class header, is the defining occurrence of the name of the class. A class name is globally visible, i.e. visible in all other classes in the specification. Therefore, semantic rules dictate that class names must be unique throughout the specification.
Also, all constructs defined within a class must have a unique name, e.g. it is not allowed to define an operation and a type with the same name.
This demand, however, does not guarantee that name conflicts will not arise. For instance, it is possible that two constructs of the same kind and with the same name are inherited from different superclasses. This kind of name conflicts must explicitly be resolved through *name qualification*, i.e. prefixing the construct with the name of the superclass and a ‘-sign (back quote).

### 3.3.3 Instance variable definitions

An object, instantiated from a class description, has an internal or global state, also called the *instance variables* of that object. Instance variables are primarily defined by the following rules:

Listing 3.2: Instance variable definitions

```
1   instance variable definitions = 'instance', 'variables',
2       [ instance variable definition ,
3       { ';', instance variable definition } , [ ';' ] ] ;
4   instance variable definition =
5     access assignment definition
```

```
6  | invariant definition ;
7  access assignment definition =
8          (['access'] , ['static'])
9        | (['static'] , ['access']) ,
10       assignment definition ;
11 assignment definition = identifier , ':' , type ,
12       [ ':=' , expression ] ;
13 invariant definition = 'inv' , expression ;
```

The section describing the internal state of a class is preceded by the keyword `instance variables`. A list of instance variable definitions and/or invariant definitions follows. Each instance variable definition consists of instance variable name with its corresponding type indication and an optional value. Furthermore, it is possible to restrict the values of the instance variables by means of invariant definitions. The overall invariant expression of a class consists of all the invariant definitions of the class and its superclasses defined as a logical `and` in the order that they are defined in.

### 3.3.4 Type definitions

As in traditional programming languages, it is possible to define data types in OML and give them appropriate names. Types can be constructed following the guidelines below:

Listing 3.3: Type definitions

```
1  type definitions = 'types' ,
2       [ access type definition ,
3         { ';' , access type definition } ,
4         [ ';' ]
5       ] ;
6  access type definition =
7          (['access'] , ['static'])
8        | (['static'] , ['access']) ,
9       type definition ;
10 type definition = identifier , '=' , type ,
11      [ invariant ]
12 | identifier , '::' ,  field list ,
13      [ invariant ] ;
```

```
14 field list = { field } ;
15 field = [ identifier , ':' ] , type
16 | [ identifier , ':-' ], type ;
```

In Overture, a type definition associates a type name with a type. To do so is considered good practice in general, but it is especially useful in situations where the defined type is needed in multiple places. Writing a type definition then assures that, whenever the definition of a type is changed, this change only needs to be made at one place in the model. Moreover, if the data type might contain values which should not be allowed, then it is possible to restrict a type by means of an invariant. Thus, by means of a predicate the acceptable values of the defined type are limited to those for which this expression evaluates to *true*. It should be noted that these invariant definitions are not part of the overall invariant expression of a class.

### 3.3.5 Function and operation definitions

In OML, algorithms can be defined by both functions and operations. What separates functions from operations is the use of local and global variables. Operations can manipulate both the global variables and any local variables. Functions, on the other hand, are pure in a sense that they cannot access global variables and they are not allowed to define local variables. Thus, functions are purely applicative while operations are imperative.
Another, less obvious, difference between functions and operations is that an algorithm definition for a function is called an expression, while for an operation it is called a statement.
Both constructs can use pre- and post-conditions in order to define their behaviour. A pre-condition is a truth-valued expression which specifies what must hold before the function/operation is evaluated. A pre-condition can only refer to parameter values and global variables (if it is an operation).
A post-condition also is a truth valued expression. However, as opposed to a pre-condition, a post-condition specifies what must hold after the function/operation is evaluated. Also, a post-condition can refer to the result identifier, the parameter values, the current values of global variables and the old values of global variables (which is the value as it was before the operation was evaluated). Note that only within operations it is possible to reference the old values of global variables in a post-condition, as functions are not allowed to change the global variables.

#### 3.3.5.1 Operation definitions

The syntax for defining an operation is given below:

Listing 3.4: Operation definitions

```
operation definitions = 'operations',
    [ access operation definition,
      { ';', access operation definition } ,
      [ ';' ]
    ] ;
access operation definition =
       (['access'] , ['static'])
     | (['static'] , ['access']) ,
    operation definition ;
operation definition = explicit operation definition
| extended explicit operation definition ;
explicit operation definition = identifier, ':',
    operation type, identifier, parameters, '==',
    operation body,
    [ 'pre', expression ], [ 'post', expression ] ;
extended explicit operation definition = identifier,
    parameter types, [ identifier type pair list ],
    '==', operation body, [ externals ],
    [ 'pre', expression ], [ 'post', expression ],
    [ exceptions ] ;
```

#### 3.3.5.2 Function definitions

It is possible to make a distinction between first and higher order functions. A higher order function is either a Curried function (a function that returns a function as result), or a function that takes one or more functions as arguments. Additionally, first and higher order functions can be polymorphic. This means that it is possible to create generic functions that can be used on values of several different types. In general, the syntax for the definition of a function is:

Listing 3.5: Function definitions

```
function definitions = 'functions',
    [ access function definition ,
      { ';' , access function definition } ,
      [ ';' ]
    ] ;
access function definition =
       (['access'] , ['static'])
     | (['static'] , ['access']) ,
    function definition ;
function definition = explicit function definition
| extended explicit function definition ;
explicit function definition = identifier ,
    [ type variable list ] , ':' , function type ,
    identifier , parameters list , '==' ,
    function body , [ 'pre' , expression ] ,
    [ 'post' , expression ] ;
extended explicit function definition = identifier ,
    [ type variable list ] ,parameter types ,
    identifier type pair list , '==' , function body ,
    [ 'pre' , expression ] , [ 'post' , expression ] ;
```

### 3.3.6 Thread definitions

In the Overture Modeling Language concurrent systems can be modeled using threads. A thread represents an independent sequence of computations. The syntax for defining a thread is defined as follows:

Listing 3.6: Thread definitions

```
thread definitions = 'thread', [ thread definition ] ;
thread definition = periodic thread definition
| procedural thread definition ;
periodic thread definition = periodic obligation ;
periodic obligation = 'periodic' ,
    '(' , numeral , ')' , '(', name , ')' ;
procedural thread definition = statement ;
```

The periodic thread definition can be regarded as the implicit way of describing the activities in a thread. This is because a periodic thread does not actually define an algorithm that will be executed. Rather, it defines a periodic execution of an operation. A thread with a periodic obligation invokes the mentioned operation at the beginning of each time interval with the length defined by numeral.
The statement in a procedural thread definition, on the other hand, is executed sequentially, and when execution of the statement is complete, the thread dies. Synchronization between multiple threads is achieved using permission predicates, which are described in the next section.

### 3.3.7 Sync definitions

In general a complete Overture model contains objects of a passive nature (which only react when their operations are invoked) and active objects which 'breathe life' into the system. These active objects have their own processing thread of control and after start up they do not (necessarily) need interaction with other objects to continue their activities. In other words, a system could be described as consisting of a number of active clients requesting services of passive or active servers. In such a parallel environment the server objects need synchronization control to be able to guarantee internal consistency, to be able to maintain their state invariants.

The following default synchronization rules for each object apply:

- operations are to be viewed as though they are atomic, from the point of the caller;
- within an object, general internal concurrency amongst operation executions is assumed, and
- operation invocations have the semantics of a rendez-vous in case two active objects are involved. Thus if an object O1 calls an operation o in object O2, and if O2 is currently unable to start operation o, then O1 blocks until the operation may be executed. In this way, invocation occurs when both the calling object and the called object are ready.

The synchronization definition blocks of the class description provide the user with ways to override the default mutual-exclusiveness. The syntax of these blocks is given below:

Listing 3.7: Sync definitions

```
1  synchronization definitions =
2      'sync' , [ synchronization ] ;
3  synchronization = permission predicates ;
4  permission predicates = permission predicate ,
5      { ';' permission predicate } ;
6  permission predicate =
7    'per' , name , '=>' , expression
8  | mutex predicate ;
9  mutex predicate = 'mutex' , '(' , 'all' , ')'
10 | 'mutex' , '(' , name list , ')' ;
```

Permission to accept execution of a requested operation depends on a guard condition in a permission predicate of the form:

***per*** *operation name* => *guard condition*

The use of implication to express the permission means that truth of the guard condition (expression) is a necessary but not sufficient condition for the invocation. The permission predicate is to be read as stating that if the guard condition is false then there is no permission. Expressing the permission in this way allows further similar constraints to be added without risk of contradiction through inheritance for the subclasses.
A mutex predicate allows the user to specify either that all operations of the class are to be executed mutually exclusive, or that a list of operations are to be executed mutually exclusive. Operations that appear in one mutex predicate are allowed to appear in other mutex predicates as well, and may also be used in the usual permission predicates (note that it is only permitted to have one permission predicate for each operation).

### 3.3.8 Statements and expressions

Statements are used in the bodies of operations to specify an algorithm, typically affecting the instance variables in a class. A statement can be executed, as a result of which the value of instance variables can change.

Listing 3.8: Statements

```
statement = let statement | let be statement
| block statement | assign statement | if statement
| cases statement | sequence for loop | set for loop
| index for loop | while loop | nondeterministic
  statement | call statement | specification
  statement | select statement | start statement
| startlist statement | return statement | always
  statement | trap statement | recursive trap
  statement | exit statement | error statement
| identity statement ;
```

Expressions are used in various parts of the Overture language to describe a calculation that is free of side effects (meaning that an expression can never affect the value of an instance variable unless it contains a call to an operation). An expression can be evaluated by replacing the identifiers used in the expression by actual values. The evaluation of an expression results in a single value.

Listing 3.9: Expressions

```
expression = bracketed expression | let expression
| let be expression | if expression | cases
  expression | unary expression | binary expression
| quantified expression | iota expression | set
  enumeration | set comprehension | set range
  expression | sequence enumeration | sequence
  comprehension | subsequence | map enumeration | map
  comprehension | tuple constructor | record constructor
| record modifier | apply | field select | function
  type instantiation | lambda expression | new expression
| self expression | is expression | undefined expression
| isofbaseclass expression | isofclass expression
| samebaseclass expression | sameclass expression
| act expression | fin expression | active expression
| req expression | waiting expression | name
| old name | symbolic literal ;
```

### 3.3.9 Types

As with most specification and programming languages, the concept of a data type, usually referred to simply as a type, is central to OML.

Listing 3.10: Types

```
type = bracketed type | basic type | composite type
| union type | product type | optional type | set type
| seq type | map type | partial function type
| type name | type variable ;
```

In the Overture Modeling Language, types fall into two categories:

- Basic types, which simply model collections of values, and
- Constructed types, which are types resulting from applying a type constructor to other types (either constructed or basic).

Because the Overture language is primarily a modeling language rather than a programming system, data types may be unbounded. For example, there is no built-in maximum integer value in the OML int type. The individual values within types are, however, always finite. For example, strings of characters are always finite in length.
One last point about the type system of OML, which is worth mentioning at this point, is that equality and inequality can be used between any pair of values. This is in contrast to other programming languages which usually require that the operands are of the same type. Because of a construct called a union type this is not the case for OML.

# Chapter 4

# The Overture compiler

Having seen the basic layout of the Overture Modeling Language, it is now possible to start the implementation of a compiler for this language. This chapter discusses some implementation issues that arose during the construction of this part of the tool set. Much of the discussion is focused around writing a compiler using a Java-based compiler generator.

## 4.1 Introduction

Writing a compiler can be a very complicated task. To alleviate some of the difficulty that arises when creating such a complex program, compiler generators partially take over this task by generating code for lexical, syntax analysis and often also for the semantic analysis and code generation phases. Well known examples of such tools are Lex/Flex and Yacc/Bison. During the research phase of this project, many of such tools were examined. Because of the programming language that would be used during the Overture project (i.e. Java), traditional implementations of Lex and Yacc were not a real option. However, as it happens with many popular tools, conversions of these programs to Java exist. For (F)Lex this is J(F)Lex. An alternative for Yacc in Java goes by the name CUP.
On the other hand, entirely new tools exist that have been built from scratch using Java. Examples of these are SableCC and JavaCC. Next to that, it appeared that these two tools had a feature that is not available in for

instance JFlex and CUP. Both SableCC and JavaCC possess functionality for generating an out-of-the-box syntax tree and even code for walking through this tree structure (using the Visitor design pattern [Gam96]).
This last observation (a more elaborate motivation can be found in [Spe04]), among other things, led to the conclusion that either JavaCC or SableCC would be the best choice for this project. From these two, initially SableCC was chosen based upon the well-designed framework.

However, theory and practice proved to be two entirely different things. Although SableCC has a better design than JavaCC, the implementation of JavaCC proved to be much more efficient. Where SableCC needed several hours (!) in order to come up with a compiler (more often than not running out of memory, causing the process to stop altogether), JavaCC needed just a couple of minutes on the same system. Based on these observations, JavaCC was chosen afterall as the compiler generator to be used for this project.

The next paragraph will shortly explain how to create a compiler using JavaCC. Also, the construction of a parse tree using JTB will be discussed. Thereafter, the implementation of the compiler for the Overture Modeling Language will be discussed.

## 4.2 Parser generation using JavaCC and JTB

### 4.2.1 JavaCC

JavaCC is a parser generator and a lexical analyzer generator. Parsers and lexical analyzers are software components for dealing with input of character sequences. Compilers and interpreters incorporate lexical analyzers and parsers to decipher files containing programs, however lexical analyzers and parsers can be used in a wide variety of other applications as well.
So what do lexical analyzers and parsers do? Lexical analyzers can break a sequence of characters into subsequences called tokens and classify these tokens. The parser then analyzes the sequence of tokens to determine the structure of the program. In compilers, the parser usually produces a tree representing the structure of the program. This tree then serves as an input to components of the compiler responsible for analysis and code generation. By default, however, JavaCC does not generate a tree structure. For this

purpose, another tool has to be used. This tool, called Java Tree Builder (JTB), will be discussed later on in this paragraph.
JavaCC employs one single grammar file, which contains all the necessary information to generate the lexical analyzer (or Token Manager as it is called in JavaCC) and the parser. A small example of such a file is shown in listing 4.1.

Listing 4.1: JavaCC grammar file

```
1  options {
2    OPTIMIZE_TOKEN_MANAGER = true;
3  }
4
5  PARSER_BEGIN(Simple)
6   import java.io.*;
7
8   public class Simple
9   {
10    public static void main(String args[])
11    {
12      Simple parser =
13        new Simple(
14          new FileReader(args[0])
15        );
16
17      parser.start();
18    }
19  }
20 PARSER_END(Simple)
21
22 SKIP :
23 {
24    " "
25 | "\t"
26 | "\n"
27 | "\r"
28 | "\f"
29 }
30
31 TOKEN :
32 {
33    < DIGIT : ["0"-"9"] >
34 }
```

```
35
36 void start() : {int result;}
37 {
38   result=expr()
39   {System.out.println("result="+result);}
40   ";" <EOF>
41 }
42
43 int expr() : {int result; int temp;}
44 {
45   result=mexpr()
46   (temp=expr_opt() {result+=temp;} )*
47   {return result;}
48 }
49 int expr_opt() : {int result;}
50 {
51   "+"
52   result=mexpr()
53   {return result;}
54 }
55
56 int mexpr () : {int result;int temp;}
57 {
58   result=atom()
59   (temp=mexpr_opt() {result*=temp;} )*
60   {return result;}
61 }
62 int mexpr_opt() : {int result;}
63 {
64   "*"
65   result=atom()
66   {return result;}
67 }
68
69 int atom() : {Token digit;}
70 {
71   digit=<DIGIT>
72   {return Integer.parseInt(digit.toString());
73 }
```

The example in listing 4.1 shows the grammar of a very simple calculator, which can only perform additions and multiplications. An example of valid

input is: $3 + 4 * 5;$. This would give the following output: $result = 23$.
The grammar file starts with a list of options (which is optional). In this case only one option is provided, which causes the token manager to be optimized. Many other options are available, including options for debugging (see [JGF]).
Between the PARSER_BEGIN and PARSER_END constructs is a regular Java compilation unit (a compilation unit in Java lingo is the entire contents of a Java file). This may be any arbitrary Java compilation unit as long as it contains a class declaration whose name is the same as the name of the generated parser ("Simple" in the above example). Hence, in general, this part of the grammar file looks like:

```
1  PARSER_BEGIN(parser_name)
2  ...
3  class parser_name ... {
4      ...
5  }
6  ...
7  PARSER_END(parser_name)
```

JavaCC does not perform detailed checks on the compilation unit, so it is possible for a grammar file to pass through JavaCC and generate Java files that produce errors when they are compiled.
The generated parser file contains everything in the compilation unit and, in addition, contains the generated parser code that is included at the end of the parser class. For the above example, the generated parser will look like:

```
1  ...
2  class parser_name ... {
3      ...
4      // generated parser is inserted here.
5  }
6  ...
```

The generated parser includes a public method declaration corresponding to each non-terminal in the grammar file. Parsing with respect to a non-terminal is achieved by calling the method corresponding to that non-terminal. Unlike Yacc, there is no single start symbol in JavaCC - one can parse with respect

to any non-terminal in the grammar.
The description of the parser is followed by the definitions of the tokens that the token manager should be able to recognize. Although the example only specifies DIGIT, one can imagine that more complex grammars would include, for instance, keywords of the language that the parser should be able to recognize.
Finally, the productions for recognizing the actual language are specified. These productions can employ BNF-like notations for specifying optional or multiple occurrences:

- "(...)*": the expression enclosed within the parentheses can occur zero or more times;
- "(...)+": the expression enclosed within the parentheses can occur one or more times;
- "(...)?": the expression enclosed within the parentheses is optional (i.e. can occur once or not at all). This can also be written as "[...]";
- "...|...": represents a choice between two (or possibly more) alternatives.

These productions can be interleaved with Java code. In the example, the code evaluates the input and calculates the result. However, more complex code can also be provided, for instance for creating a syntax tree. Although this can entirely be done by hand, tools are available to do this automatically. One such tool is the Java Tree Builder or JTB for short.

### 4.2.2 JTB

An example of a JTB grammar file (which recognizes the same grammar as the grammar file for JavaCC shown earlier) is shown in listing 4.2. JTB makes use of the same grammar as JavaCC does. There are, however, a few constraints. First, all productions should have a return type void. Additionally, the use of Java code is discouraged, because it might conflict with the code generated by JTB.

Listing 4.2: JTB grammar file

```
1  options {
2    OPTIMIZE_TOKEN_MANAGER = true;
3  }
4
5  PARSER_BEGIN(Simple)
6   import java.io.*;
7
8   public class Simple
9   {
10    public static void main(String args[])
11    {
12      Simple parser =
13        new Simple(
14          new FileReader(args[0])
15        );
16      Node root = parser.start();
17      root.accept(new Calc());
18    }
19  }
20 PARSER_END(Simple)
21
22 SKIP :
23 {
24   " "
25 | "\t"
26 | "\n"
27 | "\r"
28 | "\f"
29 }
30
31 TOKEN :
32 {
33   < DIGIT : ["0"-"9"] >
34 }
35
36 void start() : {}
37 {
38   expr() ";" <EOF>
39 }
40
41 void expr() : {}
42 {
```

```
43    mexpr() (expr_opt())*
44  }
45  void expr_opt() : {}
46  {
47    "+" mexpr()
48  }
49
50  void mexpr () : {}
51  {
52    atom() (mexpr_opt())*
53  }
54  void mexpr_opt() : {}
55  {
56    "*" atom()
57  }
58
59  void atom() : {}
60  {
61    <DIGIT>
62  }
```

From this grammar file, JTB generates (among other things) the grammar file shown in listing 4.3. This file can then be processed as any 'normal' grammar file by JavaCC, which generates the parser.

Listing 4.3: JavaCC grammar file generated by JTB.

```
1  //
2  // Generated by JTB 1.2.2
3  //
4
5  options {
6    OPTIMIZE_TOKEN_MANAGER = true;
7  }
8
9  PARSER_BEGIN(Simple)
10   import java.io.*;
11   import syntaxtree.*;
12   import java.util.Vector;
13
14   public class Simple
15   {
```

```
16     public static void main(String args[])
17     {
18       Simple parser =
19         new Simple(
20           new FileReader(args[0])
21         );
22       Node root = parser.start();
23       root.accept(new Calc());
24     }
25   }
26
27   class JTBToolkit {
28     static NodeToken makeNodeToken(Token t) {
29       return new NodeToken(
30         t.image.intern(),t.kind,t.beginLine,
31         t.beginColumn,t.endLine,t.endColumn
32       );
33     }
34   }
35 PARSER_END(Simple)
36
37 SKIP :
38 {
39   " "
40 | "\t"
41 | "\n"
42 | "\r"
43 | "\f"
44 }
45
46 TOKEN :
47 {
48   <DIGIT: ["0"-"9"]>
49 }
50
51 start start() :
52 {
53   expr n0 = null;
54   NodeToken n1 = null;
55   Token n2 = null;
56   NodeToken n3 = null;
57   Token n4 = null;
58 }
```

```
59  {
60     n0=expr()
61     n2=";" { n1 = JTBToolkit.makeNodeToken(n2); }
62     n4=<EOF> {
63        n4.beginColumn++; n4.endColumn++;
64        n3 = JTBToolkit.makeNodeToken(n4);
65     }
66
67     { return new start(n0,n1,n3); }
68  }
69
70  expr expr() :
71  {
72     mexpr n0 = null;
73     NodeListOptional n1 = new NodeListOptional();
74     expr_opt n2 = null;
75  }
76  {
77     n0=mexpr()
78     (
79        n2=expr_opt()
80        { n1.addNode(n2); }
81     )*
82     { n1.nodes.trimToSize(); }
83
84     { return new expr(n0,n1); }
85  }
86
87  expr_opt expr_opt() :
88  {
89     NodeToken n0 = null;
90     Token n1 = null;
91     mexpr n2 = null;
92  }
93  {
94     n1="+" { n0 = JTBToolkit.makeNodeToken(n1); }
95     n2=mexpr()
96
97     { return new expr_opt(n0,n2); }
98  }
99
100 mexpr mexpr() :
101 {
```

```
102     atom n0 = null;
103     NodeListOptional n1 = new NodeListOptional();
104     mexpr_opt n2 = null;
105  }
106  {
107     n0=atom()
108     (
109        n2=mexpr_opt()
110        { n1.addNode(n2); }
111     )*
112     { n1.nodes.trimToSize(); }
113
114     { return new mexpr(n0,n1); }
115  }
116
117  mexpr_opt mexpr_opt() :
118  {
119     NodeToken n0 = null;
120     Token n1 = null;
121     atom n2 = null;
122  }
123  {
124     n1="*" { n0 = JTBToolkit.makeNodeToken(n1); }
125     n2=atom()
126
127     { return new mexpr_opt(n0,n2); }
128  }
129
130  atom atom() :
131  {
132     NodeToken n0 = null;
133     Token n1 = null;
134  }
135  {
136     n1=<DIGIT> { n0 = JTBToolkit.makeNodeToken(n1); }
137
138     { return new atom(n0); }
139  }
```

Also, JTB generates a class for every production in the grammar. These classes will be used as nodes in the syntax tree. In order to walk through this tree, JTB generates two visitors, which visit every node in the tree. The

programmer, therefore, only has to override those methods for which specific actions must be performed.
One final note on the code generated by JTB concerns the handling of special constructs like "(...)*" or "[...]". For these constructs, six general classes are generated. The classes and a short description of each of them, are shown in table 4.1.

| Class name | Description |
|---|---|
| Node | Node interface that all tree nodes implement. |
| NodeListInterface | List interface that NodeList, NodeListOptional, and NodeSequence implement. |
| NodeChoice | Represents a grammar choice such as ( A \| B ). |
| NodeList | Represents a list such as ( A )+. |
| NodeListOptional | Represents an optional list such as (A )*. |
| NodeOptional | Represents an optional such as [ A ] or ( A )?. |
| NodeSequence | Represents a nested sequence of nodes such as ( A B C )+. |
| NodeToken | Represents a token string such as "class". |

Table 4.1: Automatically-Generated Tree Node Interface and Classes [JTB].

## 4.3 JavaCC/JTB grammar for OML

In this paragraph, the grammar parts as they have been discussed in paragraph 3.3, are converted into their JavaCC counterparts. Some small modifications had to be made in order for the grammar to be more easily recognized by JavaCC. An overview of all the changes, along with the entire JavaCC grammar, can be found in appendix B. Also, some of the changes will be treated in the next few paragraphs.

### 4.3.1 Documents and classes

Listing 4.4 shows the JavaCC/JTB definition for the non-terminals *document* and *class* (called "klasse", which is the dutch word for "class", to avoid name clashes with the Java keyword "class").

Listing 4.4: Documents and classes

```
1  void document() : {}
2  {
3    (klasse())+
4    <EOF>
5  }
6
7  void klasse() : {}
8  {
9    "class" <IDENTIFIER> [inheritance_clause()]
10   [class_body()]
11   "end" <IDENTIFIER>
12 }
13 void inheritance_clause() : {}
14 {
15   "is" "subclass" "of" <IDENTIFIER> (more_parents())*
16 }
17 void more_parents() : {}
18 {
19   "," <IDENTIFIER>
20 }
```

As can be seen in listing 4.4, the definition of documents and classes is fairly straightforward. However, this listing shows something important. Remember that JTB transforms nested sequences into a somewhat anonymous node called "NodeSequence". The reason for this is that it is hard to retrieve return values from such nodes. Internally, processing a NodeSequence means that several calls are made to another method of which the return values are stored in a list structure (a Vector in this case). In the worst case, this means that this Vector contains a Vector which contains a Vector etc. This makes retrieving an actual return value very difficult.
Although this might not seem to be a big problem at first, it proved to make traversing the syntax tree and especially carrying out actions at certain points very cumbersome. Therefore, the use of nested sequences of nodes has been avoided as much as possible. However, this comes at a small cost. An extra production is necessary where normally a nested sequence of nodes would occur. An example is the productions "more_parents" in listing 4.4.

### 4.3.2 Definition blocks

Because in paragraph 3.3 each of the definition blocks has been discussed separately, it is not necessary to do so again at this point. The JavaCC productions very much resemble the ones shown earlier. As an example only the operation definitions are shown in listing 4.5.

Listing 4.5: Operation definitions

```
1  void operation_definitions() : {}
2  {
3    "operations" [operations_body()]
4  }
5
6  void operations_body() : {}
7  {
8    access_operation_definition() ";"
9    (access_operation_definition_opt())*
10 }
11 void access_operation_definition_opt() : {}
12 {
13   access_operation_definition() ";"
14 }
15
16 void access_operation_definition() : {}
17 {
18   [modifiers()] operation_definition()
19 }
20
21 void operation_definition() : {}
22 {
23   <IDENTIFIER>
24   (
25   explicit_operation_definition()
26   |
27   extended_explicit_operation_definition()
28   )
29 }
30
31 void explicit_operation_definition() : {}
32 {
33   ":" operation_type() <IDENTIFIER>
```

```
34    parameters() "=="
35    operation_body() [pre()] [post()]
36  }
37
38  void extended_explicit_operation_definition() : {}
39  {
40    parameter_types() [identifier_type_pair_list()]
41    "==" operation_body() [externals()]
42    [pre()] [post()] [exceptions()]
43  }
```

Although at first glance, listing 4.5 does not appear to look much like listing 3.4, the differences are not that large. Most of the differences are caused by the fact that JavaCC must be able to discriminate between the different kinds of operations (i.e. explicit, implicit or extended explicit). This causes the need to spread these operations over several productions.
There is, however, one important difference. If one looks closely at "operations_body" in listing 4.5 and compares this to "operation definition" in listing 3.4, one might notice that the semicolon, that is used to divide the various operations, is not used in the same way in both listings. The productions used in the JavaCC grammar demands that every operation is followed by a semicolon, even the last operation. In the original language definition of OML, the last semicolon could be omitted.
The change, with respect to the original language definition, has been made in order to make it easier for JavaCC to recognize the end of the last operation. Another, albeit less important, incentive for the change was to remove the ambiguity (from a user perspective) of where the semicolon is obligatory and where it can be omitted.
This change has been made in various other places, all of which are mentioned in appendix B.

### 4.3.3 Statements and expressions

Although statements can be translated fairly literally from the language definition of OML, expressions caused some more problems. The reason for this is not only the magnitude of the number of expressions, but especially the fact that operator precedence had to be taken into account. The precedence ordering for operators is defined using a two-level approach: operators are

divided into families, and an upper-level precedence ordering, >, is given for the families, such that if families F1 and F2 satisfy F1 > F2 then every operator in the family F1 is of a higher precedence than every operator in the family F2. The type constructors are treated separately, and are not placed in a precedence ordering with the other operators (see par. 4.3.4). There are six families of operators, namely Combinators, Applicators, Evaluators, Relations, Connectives and Constructors:

- Combinators: Operations that allow function and mapping values to be combined, and function, mapping and numeric values to be iterated.
- Applicators: Function application, field selection, sequence indexing, etc.
- Evaluators: Operators that are non-predicates.
- Relations: Operators that are relations.
- Connectives: The logical connectives.
- Constructors: Operators that are used, implicitly or explicitly, in the construction of expressions; e.g. if-then-elseif-else, '|->', '. . . ', etc.

The precedence ordering on the families is: combinators > applicators > evaluators > relations > connectives > constructors. Within each family, the operators are ordered according to their precedence value, where the lowest precedence level is 1. The six families, their operators and their precedence levels can be found in tables 4.2 - 4.6 below. Not shown in this overview is the family of constructors. This family includes all the operators to construct a value. Their priority is given either by brackets, which are an implicit part of the operator, or by the syntax.

| Precedence level | Name | Symbol |
|---|---|---|
| 1 | Logical equivalence | <=> |
| 2 | Imply | => |
| 3 | Or | or |
| 4 | And | and |
| 5 | Not | not |

Table 4.2: The family of connectives

| Precedence level | Name | Symbol |
|---|---|---|
| 1 | Relational infix operators | =, <>, <, <=, >, >= |
| | Set relational operators | subset, psubset, in set, not in set |

Table 4.3: The family of Relations.

| Precedence level | Name | Symbol |
|---|---|---|
| 1 | Arithmetic prefix operator | +, -, abs, floor |
| | Set infix operator | union, \ |
| | Map infix operator | munion, ++ |
| | Sequence infix operator | ^ |
| 2 | Arithmetic infix operator | *, /, rem, mod, div |
| | Set infix operator | inter |
| 3 | Map prefix operator | inverse |
| 4 | Map infix operator | <:, <-: |
| 5 | Map infix operator | :>, :-> |
| 6 | Unary operator | +, -, abs, floor, card, power, dinter, dunion, dom, rng, merge, len, elems, hd, tl, conc, inds |

Table 4.4: The family of Evaluators.

| Precedence level | Name | Symbol |
|---|---|---|
| 1 | Subsequence | *expression* '(' *expression* ',...,' *expression* ')' |
| | Apply | *expression* '(' [*expressionlist*] ')' |
| | Function type instantiation | *expression* '[' *type* { ',' *type* } ']' |
| | Field select | *expression* '.' *identifier* |

Table 4.5: The family of Applicators.

| Precedence level | Name | Symbol |
|---|---|---|
| 1 | Iterator | ** |

Table 4.6: The family of Combinators.

If the precedence information is applied onto the concrete syntax as it can be found in appendix A, the grammar, as shown in listing 4.6 results (only the family of connectives is shown).

Listing 4.6: The family of Connectives.

```
1  void logical_equivalence_expression() : {}
2  {
3    imply_expression()
4    (LOOKAHEAD(2) logical_equivalence_opt())*
5  }
6  void logical_equivalence_opt() : {}
7  {
8    "<=>" imply_expression()
9  }
10
11 void imply_expression() : {}
12 {
13   or_expression()
14   (LOOKAHEAD(2) imply_opt())*
15 }
16 void imply_opt() : {}
17 {
18   "=>" or_expression()
19 }
20
21 void or_expression() : {}
22 {
23   and_expression()
24   (LOOKAHEAD(2) or_opt())*
25 }
26
27 void or_opt() : {}
28 {
29   "or" and_expression()
30 }
31
32 void and_expression() : {}
33 {
34   not_expression()
35   (LOOKAHEAD(2) and_opt())*
36 }
37
```

```
38 void and_opt() : {}
39 {
40   "and" not_expression()
41 }
42
43 void not_expression() : {}
44 {
45   "not" not_expression()
46 | relational_expression()
47 }
```

From listing 4.6 it can be seen how JavaCC interprets the precedence of connectives. For instance, a logical equivalence expression consists of imply expressions which are connected by a logical equivalence operator. This is exactly what should happen based on the information of the precedence level of the logical equivalence operator. The same line of reasoning can be used for the other operators.

### 4.3.4 Types

As mentioned before, type operators have their own separate precedence ordering, which is as follows (again 1 is the lowest precedence level):

1. Function type: '->'
2. Union type: '|'
3. Product type: '*'
4. Map types: 'map' ... 'to' ... and 'inmap' ... 'to' ...
5. Unary type operators: 'seq of', 'seq1 of', 'set of'

In conformity to the way precedence handling works in expressions, the same procedure is used for types, as can be seen in listing 4.7.

Listing 4.7: Precedence ordering in types.

```
1  void type() : {}
2  {
3    union_type() (LOOKAHEAD(2) type_opt())*
4  }
5
6  void type_opt() : {}
7  {
8    "->" union_type()
9  }
10
11 void union_type() : {}
12 {
13   product_type()
14   (LOOKAHEAD(2) union_type_opt())*
15 }
16
17 void union_type_opt() : {}
18 {
19   "|" product_type()
20 }
21
22 void product_type() : {}
23 {
24   map_type()
25   (LOOKAHEAD(2) product_type_opt())*
26 }
27
28 void product_type_opt() : {}
29 {
30   "*" map_type()
31 }
32
33 void map_type() : {}
34 {
35   general_map_type()
36 | injective_map_type()
37 | unary_type()
38 }
39
40 void general_map_type() : {}
41 {
42   "map" unary_type()
```

```
43    "to" unary_type()
44  }
45
46  void injective_map_type() : {}
47  {
48    "inmap" unary_type()
49    "to" unary_type()
50  }
51
52  void unary_type() : {}
53  {
54    seq_or_set_type()
55  | primary_type()
56  }
57
58  void seq_or_set_type() : {}
59  {
60    ("seq" | "seq1" | "set")
61    "of" unary_type()
62  }
```

## 4.4 JavaCC and syntax errors

Although JavaCC has some very nice features (see [Spe04]), the way in which JavaCC deals with errors in the input is not one of them. Listing 4.8 shows an example of an input file for the compiler (the example comes from [Fit04] and was slightly modified). However, this file contains some syntax errors. Namely, on line 7 a semicolon is omitted. The second error can be found on line 11, where the token '==>' is incorrectly spelled as '== >'. The third and final error is on line 12, where the beginning of the operation body is marked by '=' which should actually be '=='.
Next, in listing 4.9, the result of the normal JavaCC compiler is shown.

Listing 4.8: Erroneous sample input

```
1  class Reflector is subclass of Configuration
2
3  instance variables
4     inv
```

```
5         next = nil and
6         dom config inter rng config = {} and
7         dom config union rng config = alph.GetIndices()
8
9   operations
10     public Reflector :
11       nat * Alphabet * inmap nat to nat ==> Reflector
12     Reflector ( psp , pa , pcfg ) =
13        is not yet specified ;
14
15  end Reflector
```

Listing 4.9: Output of normal compiler when run with input of listing 4.8

```
1  Encountered "operations" at line 9, column 1.
2  Was expecting one of: ";" , "<=>" , "=>" , "or" ,
3  "and" , "not" , "in" , "psubset" , "subset" , "<>" ,
4  "=" , ">" , ">=" , "<" , "<=" , "+" , "-" , "^" , "++" ,
5  "\\" , "munion" , "union" , "*" , "/" , "inter" , "div" ,
6  "mod" , "rem" , "<:" , "<-:" , "." , "(" , "**"
```

This behaviour is, of course, not desirable. In this case it would take three tries to remove all the errors from the input file. It would be nicer if the compiler would have some form of error repair, i.e. the ability to guess what the intended input was and continue with the result of that guess as if it was actually found in the input. Therefore it was decided to improve JavaCC in order to incorporate this feature.
The adapted version of JavaCC is indeed able to find all three errors that are present in the input show in listing 4.8. The output generated by the adapted compiler is shown in listing 4.10. The repairs are also shown in this listing and are surrounded by quotation marks.

Listing 4.10: Output of adapted compiler when run with input of listing 4.8

```
1  Errors have been encountered in test.oml:
2  - Encountered "operations" (value: operations) at line 9,
3    column 1 therefore I inserted ";" before "operations"
4    (value: operations) at line 9, column 1.
5  - Encountered "==" (value: ==) at line 11, column 62
```

```
6    therefore I inserted "==>" before <IDENTIFIER>
7    (value: Reflector) at line 11, column 67.
8  - Encountered "=" (value: =) at line 12, column 39 while
9    you probably meant "==" which I recognized at line 12,
10   column 39.
11
12 Below you can see how the parser repaired the errors:
13
14
15 class Reflector is subclass of Configuration
16
17 instance variables
18    inv
19       next = nil and
20       dom config inter rng config = {} and
21       dom config union rng config = alph.GetIndices() ";"
22
23 operations
24    public Reflector :
25      nat * Alphabet * inmap nat to nat "==>" Reflector
26    Reflector ( psp , pa , pcfg ) "=="
27       is not yet specified ;
28
29 end Reflector
```

This result was reached by using a combination of so-called *follow-set error recovery* [Gru00] and the *Burke-Fisher error repair* algorithm [Bur83] and even the *panic mode* is present as a fail-safe measure.
However, before going deeper into these two subjects and the way in which they have been used in this project, it is necessary to state the ideal results of these modifications to JavaCC.

### 4.4.1 Desired functionality for error repair

According to the previous section, a compiler should not only accept correct programs but also react appropriately to syntactically incorrect programs. The desired reactions of the parser can be classified as follows [Wil95]:

1. Report and locate the error;

2. Diagnose the error;
3. Correct the error;
4. Recover to discover more (possible) errors.

Although ideally we would like the parser to detect and repair every single error in the input, this is not always possible. Several reasons exist that explain why this is unrealistic [Wil95]. First, an error may remain unnoticed near another syntactic error. Another reason is that the token at which the error occurs might not be the actual point at which the error is located. The parser could possibly have chosen a production based upon an erroneous token. In such a case, the error is only a symptom of the presence of an error rather than the error itself.
Therefore, it would be nice if the parser would at least be able to repair some of the errors and generate a valid (although not necessarily the desired) output. This would already be a great improvement over the normal behaviour of JavaCC.

### 4.4.2 Error repair in JavaCC

As has been explained in [Spe04], a very nice error repair algorithm exists which has been developed by M. G. Burke as part of his PhD work [Bur83]. In short, this algorithm is based upon deferring actual parsing actions on tokens. These deferred tokens are temporarily stored in a queue. As soon as this queue reaches its maximum size, the parse actions are executed on the token that was added to the queue earliest. In this way, when an error occurs the parser is able to trace back its steps based upon the tokens in the queue and the repair algorithm can start at a point in the input before the token at which the actual error occurred. This strategy remedies (although not always) the problem where the error that occurred is only a symptom, but not the error itself.
This strategy, however, could not be applied to JavaCC. The reason for this can be explained with the help of listing 4.11.

Listing 4.11: Overture compiler code snippet

```
1  static final public document document()
2    throws ParseException
3  {
4     NodeList n0 = new NodeList();
5     klasse n1 = null;
6     NodeToken n2 = null;
7     Token n3 = null;
8
9     label_1:
10    while (true) {
11       n1 = klasse();
12       n0.addNode(n1);
13       switch ((jj_ntk==-1)?jj_ntk():jj_ntk) {
14       case CLASS:
15          ;
16          break;
17       default:
18          jj_la1[5] = jj_gen;
19          break label_1;
20       }
21    }
22    n0.nodes.trimToSize();
23    n3 = jj_consume_token(0);
24    n3.beginColumn++; n3.endColumn++;
25    n2 = JTBToolkit.makeNodeToken(n3);
26    {if (true) return new document(n0,n2);}
27    throw new Error(
28      "Missing return statement in function"
29    );
30 }
```

This code sample was taken from the compiler generated by the original version of JavaCC. The code in this listing corresponds to following grammar rule:

$$\mathsf{document} = \mathsf{class}, \{\mathsf{class}\} ;$$

This grammar rule expects to find the keyword 'class' (identifying the start of a class) and, regardless if this keyword is present or not, a method-call is

made to a method called *klasse*. However, if this keyword is not present, an exception is thrown, but because we already started to execute a new method it is almost impossible to step back (or "unparse") to the method *document* shown in listing 4.11, try to repair the error (for instance by inserting 'class' into the input) and again call the method *klasse*. Therefore, the backtracking-aspect of the Burke-Fisher algorithm could not be implemented.
The same line of reasoning can be applied to the part of this algorithm in which a repair candidate is tested. This would require parsing a few steps ahead and then unparse again.
This explanation already shows one of the main deficiencies of the algorithm as it has been implemented in JavaCC: if an error occurs at a choice point in the grammar, the parser most likely does not recover well from the error. This deficiency will be discussed in more detail at the end of this paragraph.

Because of this limitation of the JavaCC implementation, an alternative approach was chosen. Instead of implementing the pure Burke-Fisher error repair algorithm, only a very limited version has been used. Also elements of other algorithms, namely a form of follow-set error recovery and the so-called panic mode along with some JavaCC-specific features, are part of the approach.

The parsing algorithm of JavaCC is based upon consuming tokens and making decisions about what to do next based upon the next token in the input stream (see also listing 4.11). Therefore, the parser always has a set of one (at a non-choice point in the grammar) or more (at a choice point) tokens it expects to encounter next. An example of this principle can be found in listing 4.12.
This listing shows that a value definition starts with a pattern. Next, the kind of the next token is determined (call to jj_ntk()). The result of this call determines whether or not this value definition contains a type definition (which starts with a colon). Subsequently, an equals sign ("=") should be present and finally an expression, which gives a value definition its value.

Listing 4.12: Code for creating a value definition.

```
1 static final public value_definition value_definition()
2   throws ParseException
3 {
4     pattern n0 = null;
5     NodeOptional n1 = new NodeOptional();
```

```
6       valuedef_type n2 = null;
7       NodeToken n3 = null;
8       Token n4 = null;
9       expression n5 = null;
10
11      n0 = pattern();
12      switch ((jj_ntk==-1)?jj_ntk():jj_ntk)
13      {
14        case COLON:
15          n2 = valuedef_type();
16          n1.addNode(n2);
17          break;
18        default:
19          jj_la1[28] = jj_gen;
20      }
21      n4 = jj_consume_token(EQUALS);
22      n3 = JTBToolkit.makeNodeToken(n4);
23      n5 = expression();
24      {
25        if (true) {
26          return new value_definition(n0,n1,n3,n5);
27        }
28      }
29      throw new Error(
30        "Missing return statement in function"
31      );
32  }
```

This knowledge can, of course, be used in case none of the tokens in this set is found as the next token in the input stream. Actually, this is one of the most important aspects of the algorithm used for syntax error repair in JavaCC.

When the JavaCC generated parser tries to consume a token which is not expected, a call is made to the error repair method. The first action this method undertakes is generating a list of tokens that could be present at the current state of the parser. Based on this list, the error repair method undertakes a sequence of actions, that very much resembles the simple error recovery phase from the Burke-Fisher algorithm. First, the current token (i.e. the token that caused the error, which will be called "error token" from here on) is compared to the list of tokens. If it resembles any of these tokens (i.e. it was a misspelling), then these tokens are added to a list of possible

repair candidates.
Subsequently, a second list of possible solutions is created. Every token in the set of possible tokens is tried to see if it can be inserted before the error token. To check if this possible, a kind of follow sets are used. The follow set, for the token that could be inserted, is retrieved. Next, the contents of this are examined in order to find out if the error token is part of this set. If so, the token is considered as a repair candidate.
However, this part of the algorithm contains a serious drawback. Although these sets are called "follow sets", they are not follow sets in the way normally used in compiler theory. The reason for this is the following. The same token, say **X**, could appear in several places in the grammar. Most likely, the tokens that could follow **X** are not the same in all of these places. Ideally, the follow sets used for this part of the repair algorithm would use a different kind of follow set for **X** in different states of the parser (see also the definition in, for instance, [Aho86] or [Gru00]). However, the algorithm that generates the follow sets merges all of the follow sets for one token, thus creating one large set for a specific token. This, of course, makes the insertion part of the algorithm less accurate.
Third, the error repair algorithm tries if it is possible to simply ignore the error token. This is done by again using the follow sets. However, this time they are used to check if the token that was consumed before the error token can be followed by the token that follows the error token in the input stream. As with the insertion check, this check is also less accurate than it could ideally be, because of the inaccurate follow sets.
Finally, the different parts of the error repair algorithm are repeated at a sequence of tokens ahead in the input stream, thereby possibly ignoring this sequence of tokens. The maximum amount of tokens ahead of the error token, that are considered is 10. This number is not yet based on any real theory. In [Bur83] 25 is mentioned as a sequence that is large enough to repair most errors. However, this number is based on the ideal implementation of the Burke-Fisher algorithm. As this is not the case, it seems logical to reduce the number of tokens that are considered. Some small experiments seem to show that 10 is more than enough. However, more research should be done in this area.
These four steps result in a list of possible repair candidates. This list is examined to find the most suitable candidate. The examination process is based upon a ordering from [Bur83]. This algorithm employs the following ordering for considering a possible repair candidate from the list: misspelling, insertion, deletion.
If no repair candidates have been found (or the repair algorithm tries to make the same repair twice), the repair algorithm switches to the panic

mode. In this mode, the parser searches for a semicolon (remember that the end of every definition is marked by this token). If it finds a semicolon, the token following this token is examined. Every definition starts with a limited number of tokens (for instance every sync definition starts with 'per' or 'mutex'). If the token following the semicolon could indeed mark the start of a new definition, or perhaps even a new definition block, the parser will resume its normal behaviour from this point on. This is at the cost of not parsing (i.e. skipping) an entire definition.

### 4.4.3 Evaluation of the error repair algorithm

First, it must be noted that the repair algorithm comes at no extra cost if the input does not contain any syntax errors. In such a case, the algorithm is never started and thus does not have a negative effect on the performance of the compiler.
In the event the input does contain one or more errors, the extent to which performance is decreased is entirely dependent upon the amount of errors and the amount of input. Another factor that plays a role is how well the algorithm can repair the error. If listing 4.10 is considered, the negative effect is relatively small, because the repair algorithm is able to make the ideal repair in each of the three cases. However, in many other cases the repair algorithm does not perform very well. An example is shown in listing 4.13.

Listing 4.13: Sample input

```
1  class Reflector is subclass of Configuration
2
3  values
4     Some_Value int = 4;
5
6  instance variables
7     inv
8        next = nil and
9        dom config inter rng config = {} and
10       dom config union rng config = alph.GetIndices()
11
12 operations
13    public Reflector :
14      nat * Alphabet * inmap nat to nat ==> Reflector
```

```
15      Reflector ( psp , pa , pcfg ) =
16          is not yet specified ;
17
18  end Reflector
```

Already in this example the repair algorithm has to resort to the panic mode to recover. The reason for this is the following. Upon encountering the "int"-token in line 4, the parser was expecting an "="-token (not a ":"-token !). Therefore, the most suitable repair seems inserting the "="-token. This might seem strange, but this is exactly one of the deficiencies mentioned earlier, namely the fact that the currently used follow sets are not accurate enough for the tasks for which they are used. An accurate follow-set for the "="-token would, in this case, not contain the "int"-token. The sets that are used in the current implementation, however, do not consider this. They hold the union of all the tokens that can follow the "="-token.
Another drawback that plays a role in this poor repair is the fact that the algorithm is not able to try out the repair, because unparsing is not possible. The algorithm does find, as an alternative solution, that "int" could just be ignored (i.e. deleted from the input stream). However, this repair is never considered, because an insertion is possible. So, although a better result is possible, it is never reached because insertions are preferred over deletions. The parser is not able to refine this rigid ordering with information on whether or not the repair results in a new error.
Thus, the repair results in an input stream that is still not valid and causes the need for a new repair and eventually results in the parser having to result to the panic mode.

Although the repair algorithm is not as good as was originally intended, it does have some good properties. First, it is an improvement over the normal JavaCC behaviour in a sense that it is able to continue parsing the input in the event of an error.
Second, the current implementation leaves room for improvement. At least two improvements are possible. The easiest improvement has to do with the panic mode. As mentioned before, in this mode the parser looks for a semicolon followed by a token that indicates the start of a new definition. For example, if the algorithm switches to the panic mode inside a function or operation definition, it will start searching for a semicolon followed by one of "static", "private", "public", "protected" or an identifier. However, a semicolon can be followed by an identifier in several other places too. So the parser might be resynchronized somewhere in the the middle of that function

definition instead of at the end. On the other hand, "static", "private", "protected" and "public" can only be found at the beginning of a function definition. If the grammar of the Overture Modeling Language would be adapted such that these modifiers are no longer optional, the panic mode would be greatly improved.
A second optimization lies in the follow sets. The current implementation uses follow sets that are constructed solely for a token and the context is not considered. It is possible to make the follow sets more detailed. The problem is to retrieve the right follow set for a token when an error occurs. Currently, no solution is available that can achieve this. A solution could probably be found, but would require extensive changes to the JavaCC parser generator. Furthermore, it might even be possible to use these improved follow sets for trying out a repair candidate. If, somehow, the follow set of a token, that was found in another follow set, could be retrieved, it would thus be possible to check if the altered input stream, matches a sequence of tokens from several follow sets. This idea is roughly expressed in figure 4.1. This illustration shows the original input stream, in which a value definition is being parsed. This input, however, contains some kind of error. The repair algorithm proposes to substitute the error token by some other token.
The current algorithm will only check if the token following the error token (an equals sign in this case) can follow the token to be inserted. The improved version would go further and check if, in the current state of the parser, the '4' could follow the '=' and next if the '*' could follow the '4' and so on. If all of these checks succeed, the repair is more likely to be a good repair than is the case with the current algorithm.

## 4.5 Symbol table management

After the parser has successfully parsed the input (or at least produced something that is syntactically correct), the compiler can start to check if the input program is also semantically correct. During this process there are times when the compiler needs to obtain all known information on a given identifier. Has this proposed variable name already been declared? Does this identifier represent a type or a variable? At other points in the compilation process information will be needed to be stored in the symbol table. Have we just completed a declaration of variables? Do we need to store a value during program interpreting? All this activity suggests that a symbol table is needed to be able to, at the very least, store a wide diversity of information

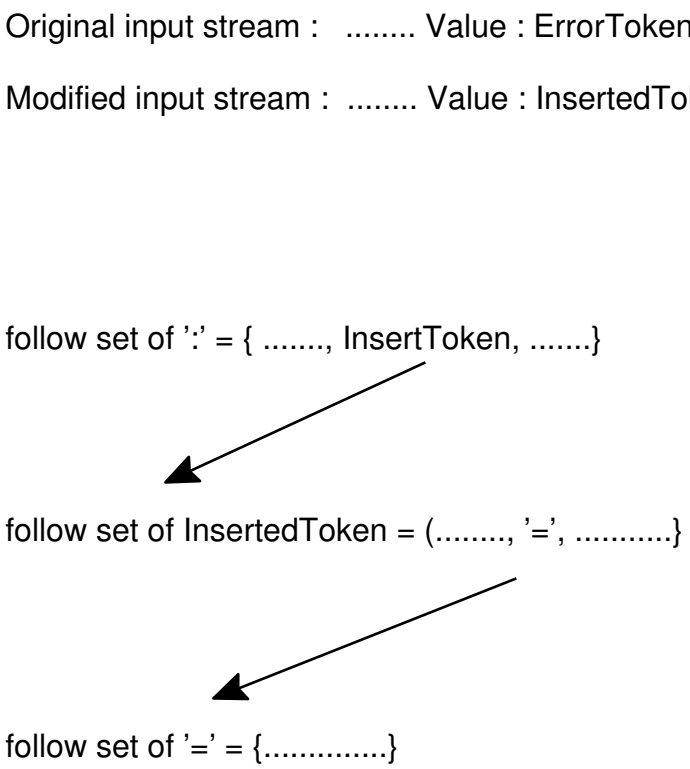


Figure 4.1: Lookahead parse check using follow sets.

and it will need to provide mechanisms for its efficient use.
Unfortunately, not much information is available on scopes, defining occurrences and applied occurrences for the Overture Modeling Language. Luckily, both Nico Plat [Pla88] and Marcel Verhoef [Ver92] have done research into related subjects. Even though their research was not on OML (or VDM++ for that matter), but on VDM-SL (in the case of Nico Plat) and on IPTES Meta-IV (in the case of Marcel Verhoef), both their works have proved very valuable.
The first part of this paragraph explores the semantic rules of OML that have to do with where variables can be declared and what their scope is (see also [Pla88] chapter 7). The second part goes more deeply into the implementation of this functionality.

### 4.5.1 Definitions

The identifiers used in an Overture specification have both an defining occurrence and one or more applied occurrences. The defining occurrence of an identifier is the place of its main, and usually only, introduction [Gru00]. This introduction supplies information about the identifier: its kind, its type, possibly an initial or fixed value etc. The other occurrences of an identifier are its applied occurrences and are the consumers of this information [Gru00]. In this section all situations are enumerated in which an identifier constitutes a defining occurrence.

#### 4.5.1.1 Classes and definition blocks

The definition of a class, and the occurrences of value, instance variable, type, function or operation definitions are considered defining. If we take a look at listing 4.13, then, based on this definition alone, it is possible to see three defining occurrences:

1. the definition of a class with the name "Reflector",
2. the definition of a value with the name "Some_Value",
3. and the definition of an operation with the name "Reflector".

Another example is the declaration of a type. The formal definition of a type has the following form:

```
1 type definition = identifier , '::',
2   field list , [invariant];
3 field list = {field};
4 field = [identifier , ':' ], type
5       | [identifier , ':-'], type;
```

Using this formal definition, a type *date* can be defined as follows:

```
types
   date :: day   : int
           month : int
           year  : int ;
```

Here *day*, *month* and *year* are considered to be defining occurrences.

#### 4.5.1.2 Variables

There are several places in an Overture specification where a programmer can introduce new variables and identifiers.
First, the let (be) statement and expression allow for new variables to be defined. So in the construct

```
let z = mk_date(31, 7, 2004) in
   a = month_domain(z)
```

The reference to *z* (after the keyword 'let') is a defining occurrence of z. In addition, block statements can be used for creating new variables. For example in

```
( dcl a := 5,
  dcl b : bool := false ;
  z := a*a ;
  b := true ;
)
```

Here, both *a* and *b* (after the keyword 'dcl') are considered to be defining occurrences of entities with identifier 'a' and 'b'.
Third, identifiers that occur in loops can be used for defining new variables. More in particular, sequence, set and index for loops. An example is shown in

```
for i = 0 to 25 do
```

```
2    is not yet specified
```

Here, $i$ is considered to be a defining occurrence.
Note that the while loop differs slightly with respect to the declaration of identifiers from other loop statements; any identifier in its control structure has an applicative rather than a defining character.
Also, if an identifier is part of a binding in a quantified expression, the occurrence of this variable identifier is considered defining.
Additionally, sequence, set and map comprehensions can contain defining occurrences of variables as shown in

```
1  { x | x:nat & x < 10 and x mod 2 = 0 }
```

Here, the first occurrence of $x$ is considered to be defining, whereas the other occurrences are applied occurrences.
Finally, cases statements and expressions can be used to introduce new variables

```
1  mergesort : seq of nat -> seq of nat
2  mergesort ( list ) ==
3      cases list :
4          []             -> [],
5          [x]            -> [x],
6          list1^list2    -> merge (
7                                mergesort ( list1 ),
8                                mergesort ( list2 )
9                            )
10     end
11 ;
```

Here both $x$, $list1$ and $list2$ are considered to be defining occurrences.

#### 4.5.1.3 Parameters

Parameter declarations occur when identifiers are part of the formal parameter list of a function or operation definition. An example is the following function definition

```
1 sqrt (r: real) res: real
2     post res**2 = r ;
```

Here, the $r$ on line 1 is the defining occurrence of this parameter. Also, $res$ is considered to be a defining occurrence, in this case the name of the variable holding the return value.

### 4.5.2 Scopes

Whether or not an identifier is "visible" within a certain part of an Overture specification is determined by the scope rules of this language. These rules not only determine if an identifier is visible, but, in the case of several identifiers with the same name, also determine which declaration of a name applies when the name appears in the program [Aho86].
OML has the following scope rules (again this information is based upon the work of Nico Plat and Marcel Verhoef):

1. Classes
   This scope begins right after the class identifier and it ends at the end of the class declaration.

2. Bodies of explicit function and operation definitions
   Such a scope begins with the parameter heading and ends at the end of the following expression (in case of a function) or the end of the following statement (in case of an operation).

3. Let (be) blocks, both in statements and expressions
   This means that in the case of the following construct:

   ```
   1 let pattern = expr1
   2   in expr2
   ```

   the scope of all defining occurrences of pattern is expr2.

4. Block statements
   The scope of a block statement begins at the opening parenthesis '(' and ends at the closing parenthesis ')'.

5. Quantified expressions
   The scope of a quantified expression begins after the corresponding keyword (i.e. one of 'forall', 'exists' or 'exists1') and ends at the end of the expression enclosed within the quantified expression.

6. Set, sequence and index for loops
   These scopes start after the opening keyword (i.e. 'for' and optionally 'all') and end at the following statement.

7. Set, sequence and map comprehensions
   The scope of these comprehensions is identified by the opening bracket '{' and the closing bracket '}'.

8. Cases statements and expressions
   Such a scope begins with the keyword 'cases' and ends at the end of the following expression (in case of a cases expression) or the end of the following statement (in case of a cases statement).

Any defined object is visible in the scope of its definition and all enclosing scopes unless it is hidden by another definition with the same identifier in an enclosed scope. A defined object is not visible in the enclosing scope or a scope parallel with the scope in which the definition occurs [Pla88].

### 4.5.3 Checking the scope rules

Checking the scope rules forms a partially integrated task with the process of creating the symbol table (i.e. identifying the defining occurrences of the variables used). In the approach chosen, a notion of the scope-hierarchy is maintained by means of a stack. Upon entering a new scope (e.g. the definition of a function), the corresponding scope is created and pushed on top of the stack. All definitions which are made from that point on until the end of the scoped definition are added to this scope.
As soon as the end of the scoped definition is reached, the scope is popped off the stack again.
An example can be found in figure 4.2. Here you can see three scopes. The first is the global scope (which is always on the stack). This scope contains the class definitions and enforces that names of classes are unique.
The second scope if opened upon entering the class definition. Within this scope, definitions of types, operations etc. can be made local to this scope.

The final scope that is created, is that of the operation definition. This scope would contain, for instance, the parameters of the operation.

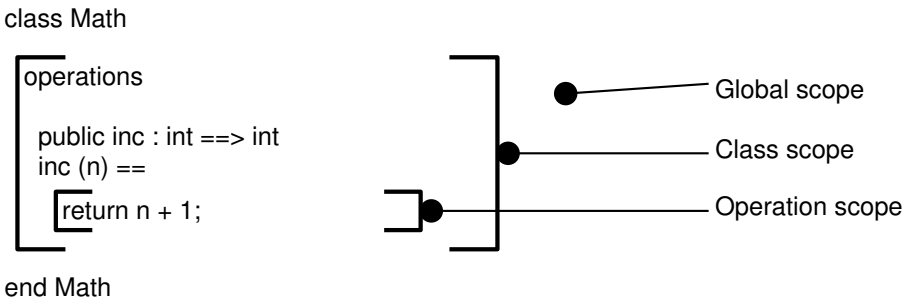


Figure 4.2: Example of scopes in a class definition.

When a defining occurrence is reached, the scope stack is searched bottom-up for a defining occurrence of an identifier with the same name. If such an identifier can be found, the compiler emits an appropriate error message notifying the user of multiple definitions of variables with the same name.
An exception is made in the case of functions and operations having the same name. When the compiler finds, for instance, two functions with the same name and the same return value, but with a different set of parameters, the compiler will not complain about this (i.e. overloading of functions and operations is allowed). Of course, when a function and an operation have the same name, this is not allowed.

One might wonder why not also applied occurrences are checked in order to determine if they have a corresponding defining occurrence. The reason for this is the following: the defining occurrence of a variable might be in a class different from the class which is currently being examined (e.g. in a parent class). The latter might not yet have been examined. In such an event, the algorithm would fail and the compiler would issue an error message even though this is probably not correct.
Therefore, finding a defining occurrence for each applied occurrence should be done in a second pass, after the creation of the entire symbol table.

# Chapter 5

# Generating output

In order to transform the original input given by the user into an XML document representing the original specification given in the Overture Modeling Language, careful thought has to be given to the structure of the XML document. This structure can be described in an XML schema document.
The first part of this chapter is devoted to creating a schema for the Overture Modeling Language. Next, in the second part, the actual creation of XML instance documents is discussed.

## 5.1 XML schema

The XML schema language of the World Wide Web Corporation (W3C) is packed with features, and there are often several ways to accurately describe the same thing. The decisions made during schema design can affect its usability, accuracy, and applicability.
Therefore it is important to keep in mind what the design objectives are when creating a schema. These objectives may vary depending on how the XML documents will be used, but some are common to all use cases. The XML schema document should accurately describe an XML instance document (or instance for short) and allow it to be validated.
Schemas should also be precise in describing data. Precision (e.g. defining restrictive data types instead of allowing a broad range of values) can result in more complete validation as well as better documentation.

Finally, schemas should be very clear, allowing a reader to understand the structure and characteristics of the instance being described, without too much effort.

Next to these three general design principles, the XML schema definition of the Overture Modeling Language also brings up some specific design issues. Of these the one with the biggest consequences has to do with the fact that the instance document is being generated by the compiler. This causes the need to design the XML schema in such a way that the compiler can easily generate an instance.
Additionally, it is important to keep in mind that the instance documents might, in later phases of the Overture project, be created manually. Thus, the schema should be easily readable and as concise as possible.
Finally, instance documents might be used to describe small parts of an Overture specification in stead of an entire document. Therefore, it must be possible to start with an arbitrary root-element in the instance document, and continue from that point on.

### 5.1.1 XML schema design approaches

The objectives mentioned above influence the way in which the XML schema is designed. Commonly, there are three approaches for designing an XML schema. These three are called Russian Doll design, Salami Slice design and Venetian Blind design [Gul02].

#### 5.1.1.1 Russian Doll design

The Russian Doll design involves nesting all of the component definitions and declarations within the schema structure itself. Everything is declared locally with the exception of the root element of the document. An example is shown in listing 5.1.

Listing 5.1: Russian Doll design.

```
1 <?xml version="1.0" encoding="ISO-8859-1" ?>
2 <xsd:schema
3     xmlns:xsd="http://www.w3.org/2001/XMLschema">
```

```
4       <xsd:element name="book">
5           <xsd:complexType>
6               <xsd:element name="title"
7                            type="xsd:string" />
8               <xsd:element name="author"
9                            type="xsd:string" />
10          </xsd:complexType>
11      </xsd:element>
12  </xsd:schema>
```

The result is a schema that may be hard to read, but resembles the corresponding XML instance documents very closely.
Next to that, the Russian Doll design has some other advantages. Because all the declarations are local in scope, and self contained, there can't be references to a declared component in another section of the schema. As a result, what you see is what you get.
Another advantage is that the namespaces are localized. If we would declare a targetNamespace for the schema in listing 4.1, and that schema would then be imported into another schema, we would only have to worry about namespace qualification with the `<book>` element, as the other elements would be local to `<book>`.
This does also have the advantage of shielding instance documents somewhat, because if the schema is included in another schema, only the root element is exposed. That means that the integrity of the element is harder to compromise.

The Russian Doll design does have a very serious disadvantage however. The reuse of components, which is one of the great features of XML schema, is entirely absent in this design principle. Not only are elements shielded when the schema is imported into another schema, they are also invisible throughout the rest of the schema.

#### 5.1.1.2 Salami Slice design

The Salami Slice design represents the other end of the design spectrum. With this design the instance document is disassembled into its individual components. In the schema each component is defined on its own (as an element declaration), and then grouped:

Listing 5.2: Salami Slice design.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xsd:schema
    xmlns:xsd="http://www.w3.org/2001/XMLschema">
     <xsd:element name="title" type="xsd:string" />

     <xsd:element name="author" type="xsd:string" />

     <xsd:element name="book">
          <xsd:complexType>
               <xsd:sequence>
                    <xsd:element ref="title" />
                    <xsd:element ref="author" />
               </xsd:sequence>
          </xsd:complexType>
     </xsd:element>
</xsd:schema>
```

An advantageous side effect is that each component declaration is clearly visible to the human reader, and easy to read. It's easy to locate and look at the structure of any one particular component, as opposed to the overall document structure.
This design also provides a great deal more flexibility when using elements, as opposed to the Russian Doll design. Global elements can be referenced from anywhere in the schema and even in other schemas when the schema is imported.

There is one major drawback to this methodology, though, when it comes to namespace use. For example, if the schema would have had a targetNamespace:

```
targetNamespace="http://www.example.com/book"
```

and the qualified namespace version would be used:

```
xmlns:book="http://www.example.com/book"
```

an instance document would need to look like the document shown in listing 5.3.

Listing 5.3: XML instance document.

```
<book:book>
    <book:title>The Hobbit</book:title>
    <book:author>J.R.R. Tolkien</book:author>
</book:book>
```

The resulting namespace clutter (especially when multiple namespaces are being used) can have an impact on readability, and can be an issue if end users aren't familiar with namespaces and namespace issues. Of course, for automatically generated schemas and instances, this is less of an issue.

#### 5.1.1.3 Venetian Blind design

This third style is often seen as the best of both worlds (i.e. the combination of the best parts of the Russian Doll design and the Salami Slice design). If the Venetian Blind design would be used for the schema defining a book, the result would look like listing 5.4.

Listing 5.4: Venetian Blind design.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xsd:schema
    xmlns:xsd="http://www.w3.org/2001/XMLschema">
    <xsd:simpleType name="title">
        <xsd:restriction base="xsd:string" />
    </xsd:simpleType>

    <xsd:simpleType name="author">
        <xsd:restriction base="xsd:string" />
    </xsd:simpleType>

    <xsd:complexType name="book">
        <xsd:sequence>
            <xsd:element name="title"
                         type="title" />
            <xsd:element name="author"
                         type="author" />
        </xsd:sequence>
    </xsd:complexType>
```

```
20
21      <xsd:element name="book" type="book" />
22  </xsd:schema>
```

The Venetian Blind design offers the component reusability, which is a strong point of the Salami Slice design, both through the global type definitions and the element definition.
It is also possible to exert a finer degree of control over namespaces, like when using the Russian Doll design, allowing the user to "toggle" off and on namespace qualification in documents using the *elementFormDefault* attribute, which by default will hide the namespaces in documents.

## 5.2 XML schema definition for OML

For an XML schema definition of the Overture Modeling Language, the Russian Doll design is not suitable. The document would get far too complex and component reuse, which is essential for keeping the document manageable, is not possible.
Also, the Salami Slice design principle is not suitable. Although it provides a high degree of reusability, it does not allow to easily hide elements from designers that want to use the schema for their own purposes. However, this might be a feature that is needed in future extensions
As a result, the Venetian Blind design pattern was used for this project. It has the desired features (i.e. component reusability and support for multiple root elements) and does not have any obvious disadvantages for the purpose it will be used for.

The remaining part of this paragraph is used to discuss the design of the schema and some of the design decisions that have been taken. The arrangement of the various sections will resemble those of par. 3.3 and par. 4.3. The entire XML schema definition can be found in Appendix C.

### 5.2.1 Documents and classes

The starting point, at least in most cases, of any XML instance document will be the document-tag. It will contain a, theoretically unbounded, list of

child elements which represent the classes in the document.
However, a specification of a real software system will generally consist of several documents. There are two ways in which the generated XML could resemble this. First, the generated XML could also be spread across several files. On the other hand, it is possible to generate just one XML file, which in turn contains several documents.
To account for the second option, the XML schema defines an element called *project*. A project-element can contain one or more document-elements. Both XML schema definitions are shown in listing 5.5, as well as the definition for a class.

Listing 5.5: Project and document and class elements.

```
<xsd:element name="project">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element maxOccurs="unbounded"
                   ref="document"/>
    </xsd:sequence>
    <xsd:attribute name="projectname"
                   type="xsd:string"
                   use="optional"/>
  </xsd:complexType>
</xsd:element>

<xsd:element name="document">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:element maxOccurs="unbounded"
                   ref="class"/>
    </xsd:sequence>
    <xsd:attribute name="filename"
                   type="xsd:string"
                   use="required"/>
    <xsd:attribute name="timestamp"
                   type="xsd:dateTime"
                   use="optional"/>
  </xsd:complexType>
</xsd:element>
```

```
30
31 <xsd:element name="class">
32   <xsd:complexType>
33     <xsd:sequence>
34       <xsd:element minOccurs="0"
35                    name="extra_information"
36                    type="extra"/>
37       <xsd:element name="identifier"
38                    type="identifier"/>
39       <xsd:element minOccurs="0" name="supercls">
40         <xsd:complexType>
41           <xsd:sequence>
42             <xsd:element maxOccurs="unbounded"
43                          name="identifier"
44                          type="identifier"/>
45           </xsd:sequence>
46         </xsd:complexType>
47       </xsd:element>
48       <xsd:group maxOccurs="unbounded"
49                  minOccurs="0"
50                  ref="definitions"/>
51     </xsd:sequence>
52   </xsd:complexType>
53 </xsd:element>
```

Although, most of what is shown in listing 5.5 is a pretty straightforward translation from the BNF definition for OML, one element is not. The element with the name extra_information is not directly part of either the BNF specification or the JavaCC grammar.
In order to understand the extra_information element, it is important to know what it looks like. The structure of this element is shown in listing 5.6.

Listing 5.6: The extra information element.

```
1 <xsd:complexType name="extra">
2    <xsd:sequence>
3       <xsd:element name="comment"
4                    type="xsd:string"
5                    minOccurs="0"/>
6       <xsd:element name="location" minOccurs="0">
7          <xsd:complexType>
8             <xsd:attribute name="beginline"
```

```
9                                   type="xsd:int" />
10              <xsd:attribute name="endline"
11                                  type="xsd:int" />
12              <xsd:attribute name="begincolumn"
13                                  type="xsd:int" />
14              <xsd:attribute name="endcolumn"
15                                  type="xsd:int" />
16          </xsd:complexType>
17        </xsd:element>
18    </xsd:sequence>
19 </xsd:complexType>
```

In any Overture specification, anywhere in the code comments can occur. It is desirable to enable other programs that rely upon the XML instance document(s), to use these comments. Therefore, comments can be stored inside the XML instance document(s) as part of the extra_information-element. Also, the extra_information-element can contain specific information on the position of the parent element in the original document.

### 5.2.2 Definition blocks

As was the case with the translation of the various definition blocks into a grammar for JavaCC/JTB, the translation to an XML schema definition was fairly straightforward. The result for the operation definitions can be seen in listing 5.7.

Listing 5.7: Operation definitions.

```
1  <xsd:group name="operationdefinition">
2    <xsd:choice>
3      <xsd:element ref="explopdef" />
4      <xsd:element ref="extexplopdef" />
5    </xsd:choice>
6  </xsd:group>
7
8  <xsd:element name="explopdef">
9    <xsd:complexType>
10     <xsd:sequence>
11       <xsd:element minOccurs="0"
```

```
12                      name="extra_information"
13                      type="extra"/>
14        <xsd:element name="identifier"
15                      type="identifier"/>
16        <xsd:element name="operationtype"
17                      type="operationtype"/>
18        <xsd:group maxOccurs="unbounded"
19                   minOccurs="0"
20                   ref="pattern"/>
21        <xsd:group ref="operationbody"/>
22        <xsd:element minOccurs="0"
23                      name="precondition">
24          <xsd:complexType>
25            <xsd:sequence>
26              <xsd:element minOccurs="0"
27                           name="extra_information"
28                           type="extra"/>
29              <xsd:group ref="expression"/>
30            </xsd:sequence>
31          </xsd:complexType>
32        </xsd:element>
33        <xsd:element minOccurs="0"
34                      name="postcondition">
35          <xsd:complexType>
36            <xsd:sequence>
37              <xsd:element minOccurs="0"
38                           name="extra_information"
39                           type="extra"/>
40              <xsd:group ref="expression"/>
41            </xsd:sequence>
42          </xsd:complexType>
43        </xsd:element>
44      </xsd:sequence>
45      <xsd:attribute ref="static" use="optional"/>
46      <xsd:attribute ref="access" use="optional"/>
47    </xsd:complexType>
48  </xsd:element>
```

The schema element definitions in listing 5.7 do not need much explanation. There are, however, a few things worth discussing. First, the operation modifiers (i.e. whether a operation is public, private, protected and/or static) are modeled as attributes of an operation definition. The reason for this is

that these modifiers are attributes of an operation and should therefore be modeled as such.
The second point is that pre- and postconditions are modeled as local elements instead of global elements. This design decision comes from the fact that pre- and postconditions belong to a specific operation (or function for that matter) and it should not be possible to create a valid XML document with a precondition as root-element. Such a document would have no meaning, and thus would be useless.
The final point worth mentioning is that *operationdefinition* is modeled as a group instead of as an element or type. The use of such an element group (as it is officially called) makes it very easy to define a group that can be reused in multiple locations, without the disadvantage of adding an extra element to the instance documents. In this case, an element definition would have resulted in the following structure in an instance document:

```
1 . . .
2   <operationdefinition>
3     <explopdef>
4       . . .
5     </explopdef>
6   </operationdefinition>
7 . . .
```

The current implementation, however, results in the structure shown below:

```
1 . . .
2   <explopdef>
3     . . .
4   </explopdef>
5 . . .
```

In both cases, the same information is modeled. However, the second implementation needs less space. This way of modeling information has been used throughout the XML schema and results in considerably smaller instance documents compared to the alternative.
Another advantage is that it results in XML instance documents that are easier to read, because they are less cluttered with superfluous information.

### 5.2.3 Statements and expressions

Listing 5.8 shows a small part of the "expression"-element. Also it shows two, very similar, elements which model the "isofclass"-expression and the "isofbaseclass"-expression.

Listing 5.8: Expressions.

```
<xsd:group name="expression">
  <xsd:choice>
    . . .
    <xsd:element name="isofclassexpr"
                 type="isofclassexpr"/>
    <xsd:element name="isofbaseclassexpr"
                 type="isofbaseclassexpr"/>
    . . .
  </xsd:choice>
</xsd:group>

<xsd:complexType name="isofbaseclassexpr">
  <xsd:sequence>
    <xsd:element minOccurs="0"
                 name="extra_information"
                 type="extra"/>
    <xsd:element name="name" type="name"/>
    <xsd:group ref="expression"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="isofclassexpr">
  <xsd:sequence>
    <xsd:element minOccurs="0"
                 name="extra_information"
                 type="extra"/>
    <xsd:element name="name" type="name"/>
    <xsd:group ref="expression"/>
  </xsd:sequence>
</xsd:complexType>
```

Apart from the fact that the "isofclass"-expression and the "isofbaseclass"-expression model different kinds of expressions, they are the same. In order

to keep the number of elements in the schema within certain limits, these two expressions could be modeled as one element. Differentiation could be made by using an attribute telling which of the two is meant. This technique could be used in various other places as well, thus reducing the number of elements.
Although this might seem appealing, it has the consequence that the resulting document is less easy to read for a human reader. It is easier to find a specific element than a specific attribute.
Additionally, if future applications of the Overture XML schema definition would model separate expressions and use such an expression as a root-element in an instance document, it is more intuitive to have a separate element instead of a more general element with an attribute denoting which specific expression is modeled by the element definition.

### 5.2.4 Types

The last element definitions that will be mentioned explicitly, model the types that can be used in OML. A small excerpt from the schema is shown in listing 5.9.

Listing 5.9: Types.

```
1  <xsd:group name="type">
2    <xsd:choice>
3      . . .
4      <xsd:element name="uniontype"
5                   type="uniontype" />
6      . . .
7    </xsd:choice>
8  </xsd:group>
9
10 <xsd:complexType name="uniontype">
11   <xsd:sequence>
12     <xsd:element minOccurs="0"
13                  name="extra_information"
14                  type="extra" />
15     <xsd:group maxOccurs="unbounded"
16                minOccurs="2"
17                ref="type" />
18   </xsd:sequence>
```

```
19 </xsd:complexType>
```

As was mentioned in [Spe04], the W3C XML Schema (WXS) language has some nice features that are unique to this language and important for the schema definition for OML. One of these features is shown in listing 5.9. From the definition of a union type (see Appendix A) we know that it models an union of two or more types. This constraint can be very nicely modeled with WXS. An element definition can be extended with information on the minimum number it should occur (denoted by *minOccurs*) within a containing element. It is also possible to give an upper limit to the number of occurrences (denoted by *maxOccurs*). In this case there is no fixed upper limit, which is denoted by the value "unbounded".

## 5.3 Generating XML with Java

Currently, there are two major standard APIs for processing XML documents with Java - the Simple API for XML (SAX) and the Document Object Model (DOM) - each of which comes in several flavors. Each of these APIs has its own strengths and weaknesses. They are shortly discussed next.

### 5.3.1 DOM

The Document Object Model (DOM) is an application programming interface (API) for valid HTML and well-formed XML documents. It defines the logical structure of documents and the way a document is accessed and manipulated. In the DOM specification, the term "document" is used in the broad sense - increasingly, XML is being used as a way of representing many different kinds of information that may be stored in diverse systems, and much of this would traditionally be seen as data rather than as documents. Nevertheless, XML presents this data as documents, and the DOM may be used to manage this data.

In the DOM, documents have a logical structure which is very much like a tree [DOM]. A very simple example is shown in figure 5.1. This figure shows how an XML document is represented by a document object tree.

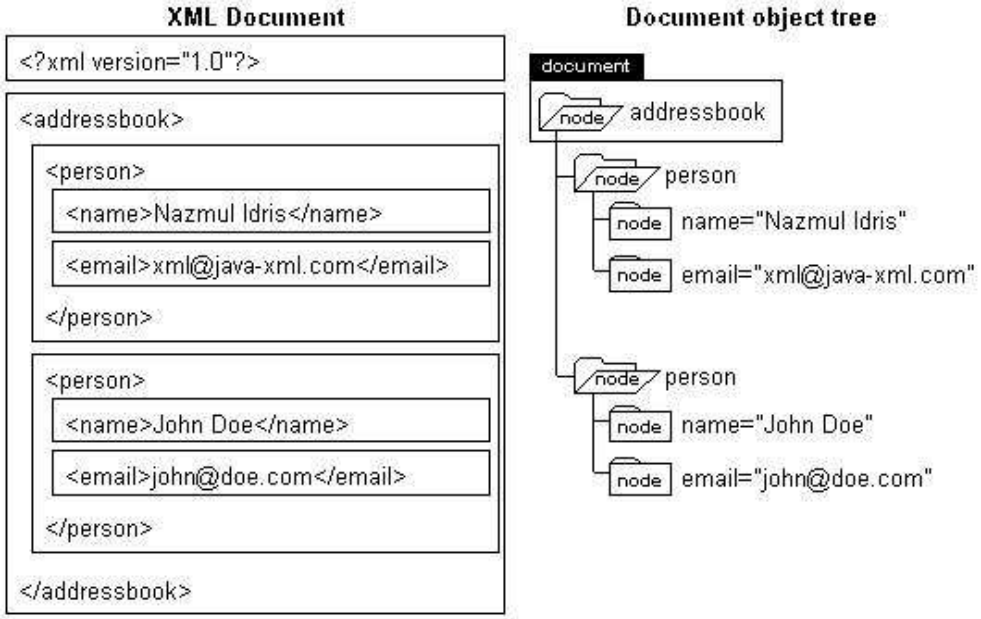


Figure 5.1: Graphical representation of a DOM tree [Idr99].

Regardless of the kind of information in an XML document (whether it is tabular data, or a list of items, or just a document), DOM creates a tree of nodes when a Document object is created, given the XML document. Thus DOM forces the programmer to use a tree model to access the information in an XML document. This works quite well because XML is hierarchical in nature. This is why DOM can put all the information in a tree (even if the information is actually tabular or a simple list).
In general, a DOM tree has a single root node, and all nodes in this tree, except for the root, have a single parent node. Furthermore, each node has a list of children (which can be empty, for instance when the node represents a leaf in the tree).
There can also be nodes that are not part of the tree structure. For example, each attribute node belongs to a single element node, but is not considered to be a child of that node. Furthermore, nodes can be removed from the tree or created but not added to the tree. Thus a full DOM document is composed of the following [Har03]:

- A tree of nodes;
- Various nodes that are somehow associated with other nodes in the tree, but are not themselves part of the tree;

- A random assortment of disconnected nodes.

The tree structure allows for easy access to all the nodes in the document. Standard ways for accessing a tree structure, for instance using recursion or design patterns like the Visitor pattern [Gam96], can be used.
Although DOM is very easy to use, it has one serious drawback. The performance of DOM implementations is very bad. Especially, applications using DOM are very space consuming. The in-memory representation of the DOM tree can be anywhere from three to ten times as large as the original document [Har03].
However, this does not necessarily mean that DOM is bad. With modern computers, speed and memory use are not a very big issue. When the XML document is very complicated, needs frequent modifications or needs to be accessed frequently (in various places), DOM is the way to go.

### 5.3.2 SAX

The SAX implementation for accessing XML documents is radically different from the DOM implementation. SAX does not use a tree or a tree-like structure, but uses events in order to access the document. Because SAX does not create a default Java object on top of the XML document, it can be (and generally is) much faster than DOM.
In the case of DOM, the parser does almost everything, it reads the XML document in, creates a Java object model on top of it and then gives a reference to this object model (a Document object) so that it can manipulated.
SAX is called the Simple API for XML because it is really simple. SAX does not expect too much from the parser. All SAX demands is that the parser reads in the XML document, and fire a bunch of events depending on what tags it encounters in the XML document. The programmer is responsible for interpreting these events by writing an XML document handler class, which is responsible for making sense of all the tag events and creating objects in some custom object model.
Although SAX has many advantages, like being fast, simple and very well suited for accessing just a small part of a document, it also has some disadvantages. With SAX it is not possible to access a document in random order. Because the document is not in memory you have to handle the data in the order it arrives.
Additionally, the design principle in SAX is that it doesn't provide the reader

with lexical information. SAX tries to tell what the writer of the document wanted to say, and avoids troubling the reader with details of the way they chose to say it. For instance, it is impossible to find out whether the original document contained "`
`" or "`
`" or whether it contained a real newline character: all three are reported to the application in the same way. As is the case with DOM, the disadvantages of SAX do not make SAX a better or worse API for accessing XML documents than DOM. It just means that SAX is sometimes better than DOM, but in other cases DOM is the API that is best suited for the job.

### 5.3.3 From OML to XML

Because of the fact that the XML instance documents will be generated by walking through the parse tree, DOM is the logical choice for generating these documents.
Several implementations of the DOM specification exist (e.g. DOM, JDOM, DOM4J), one of which is packaged, as a plugin, with Eclipse[3]. This plugin is a fully compliant XML parser. It is being developed as part of the Apache project (`http://xml.apache.org`) and is called Xerces2 for Java.
The implementation of the XML generator class, using Xerces2, has been derived from the XML schema definition shown earlier. Because the design decisions for the XML instance documents have already been described in par. 5.2 and the low level of complexity of the code itself, the XML generator class will not be discussed here. A short example of some OML and its XML counterpart is shown in listing 5.10 and 5.11.

Listing 5.10: An OML class definition.

```
1  class A
2
3  operations
4
5    public inc: int ==> int
6    inc(n) ==
7      return n + 1
8    pre n > 0;
9
10 end A
```

[3] As of Eclipse 3.0M9 the Apache Xerces plugin is no longer part of the standard distribution.

Listing 5.11: XML representation of the class from listing 5.10.

```
<?xml version="1.0" encoding="UTF-8"?>
<document timestamp="2004-07-16T12:51:30"
          filename="A.oml">
  <class>
    <identifier>
      <extra_information>
        <location beginline="1"
                  endline="1"
                  begincolumn="7"
                  endcolumn="7"/>
      </extra_information>
      <value>A</value>
    </identifier>
    <explopdef access="public">
      <identifier>
        <extra_information>
          <location beginline="5"
                    endline="5"
                    begincolumn="10"
                    endcolumn="12"/>
        </extra_information>
        <value>inc</value>
      </identifier>
      <operationtype>
        <opdom>
          <basictype>int</basictype>
        </opdom>
        <oprng>
          <basictype>int</basictype>
        </oprng>
      </operationtype>
      <patternidentifier>
        <identifier>
          <extra_information>
            <location beginline="6"
                      endline="6"
                      begincolumn="7"
                      endcolumn="7"/>
          </extra_information>
          <value>n</value>
        </identifier>
      </patternidentifier>
```

```
43          <returnstmt>
44            <binaryexpr operator="+">
45              <name>
46                <identifier>
47                  <extra_information>
48                    <location beginline="7"
49                              endline="7"
50                              begincolumn="12"
51                              endcolumn="12" />
52                  </extra_information>
53                  <value>n</value>
54                </identifier>
55              </name>
56              <numlit>
57                <extra_information>
58                  <location beginline="7"
59                            endline="7"
60                            begincolumn="16"
61                            endcolumn="16" />
62                </extra_information>
63                <value>1</value>
64              </numlit>
65            </binaryexpr>
66          </returnstmt>
67          <precondition>
68            <binaryexpr operator=">">
69              <name>
70                <identifier>
71                  <extra_information>
72                    <location beginline="8"
73                              endline="8"
74                              begincolumn="7"
75                              endcolumn="7" />
76                  </extra_information>
77                  <value>n</value>
78                </identifier>
79              </name>
80              <numlit>
81                <extra_information>
82                  <location beginline="8"
83                            endline="8"
84                            begincolumn="11"
85                            endcolumn="11" />
```

```
86                </extra_information>
87                <value>0</value>
88              </numlit>
89            </binaryexpr>
90          </precondition>
91        </explopdef>
92    </class>
93  </document>
```

Even with this short example, the XML representation becomes quite large in comparison to the original class definition. The reason for this lies partly in the fact that XML is a very verbose language by design.
Another cause of the lengthy instance document is the fact that the XML is being generated instead of hand written. Each block (for instance an expression) is treated as-is without considering context information. Therefore, XML is always generated in the same way for the same block. This results in superfluous information, but this is unavoidable without making the underlying code very complex.

# Chapter 6

# Conclusions

## 6.1 The Master's degree project

During this Master's project, an implementation of a syntax checker and a (partial) static semantics checker for the Overture Modeling Language has been realized. Also a front-end has been created using the Eclipse Platform. Finally, an XML schema for the Overture Modeling Language along with an XML generator has been implemented.
The implementation allows for easy extension with more elaborate checks of the static semantics. Also it is possible to add entirely new functionality in the form of Eclipse plugins which can extend existing plugins.

The usefulness of the current implementation is limited for several reasons. First, the current implementation of the static semantics checker is limited to checking the uniqueness of the defining occurrences of identifiers. However, the static semantics of OML encompass much more than that (see chapter 3). The current implementation should therefore be extended to incorporate a full static semantics checker. Also, sync and thread definitions have not been considered at all, due to time constraints.
Additionally, the Overture Modeling Language is still in a state of flux. This implies that the language definition is not stable, and that some properties of the language have not been defined yet. Therefore, a fully correct analyzer will only be possible after this has been worked out. Additionally, changes in the language definition will have their effects on the implementation of the editor for Eclipse.

Finally, the current implementation does not consider inheritance relationships (or any other relationship between classes for that matter) between the classes declared in a single project. Only the file in the active editor is considered. The current version has no need for these relationships, because the static semantics checker (the only part that would need this information at the moment) has not yet been implemented to such an extent that it needs this information.

## 6.2 JavaCC and JTB

During this project, extensive use has been made of two tools for generating the compiler and the syntax tree. Both tools proved to be very powerful. However, the usefulness of JTB as a tool for generating a parse tree is questionable.
The problem is that it does not make a syntax tree, that can be used "out-of-the-box" for all sorts of tasks, for instance static semantics checking or pretty printing. Although all of these tasks can be done using the syntax tree generated by JTB, it would probably be better to write code for generating a tree structure manually. In this way, the generated syntax tree will probably relate better to things like "an expression" or "a type definition". This would make the tasks, for which the syntax tree is needed, easier to execute. The current syntax tree stays to close to the original grammar, which makes it too fine-grained and overly complicated.
Another incentive for manually writing code for the syntax tree is that this makes it easier to abstract from the underlying code that has been generated by the compiler generator. If, at any time in the future, the code for the compiler will be generated by another compiler generator, this will only affect the code that generates the syntax tree. The API for the syntax tree itself will not have to change. This is, of course, a desirable course of events, because API changes in the syntax tree have a great impact on other parts of the tool set. In contrast, with the current implementation the use of another compiler generator *will* have such an effect.

## 6.3 XML and XML schema

The choice for W3C XML Schema (which has been motivated in [Spe04]) as the schema language to be used in this project, seems to have worked out

well. The schema that has been produced can easily be used for all sorts of processes, like for instance validation, and is also suited as documentation of the instance documents.
Especially this last property will probably prove to be very useful. The reason for this is that the XML instance documents themselves can get very large and are therefore hard to read (at least to humans). The schema document helps to make it easier by allowing the reader to easily grasp the structure of such instance documents.
One question does remain however. As mentioned before, the XML instance documents can get very large even for small specifications. This is partly caused by the fact that the XML language is a very verbose language by design. On the other hand, a lot of information is repeated, because of the design of the XML schema.
Of course, the first problem cannot be remedied. With respect to the second point, however, options exist to make improvements upon the current design of the XML schema. If the XML generator could take context information into account, some element definitions could be omitted. For instance, the instance document shown in listing 5.11 defines a $patternidentifier$ on line 32. The immediate child of this element is an $identifier$. This last element bares no new information and could be omitted. This trick can be exploited in several other places as well and would help reduce the size of the generated documents.
Nevertheless, the instance documents will still be much larger than the documents they are created from. So a second question arises. Is XML the way to go for the Overture project? More efficient alternatives exist, if the only purpose of the XML instance documents would be exchanging information between different parts of the tool set. However, the general idea about the use of XML is that the whole range of XML tools and related language (like for instance XMI) becomes available. With this in mind, it seems like XML is the way to go and the drawbacks of this language will just have to be taken for granted.

## 6.4 Eclipse

Last, but certainly not least, of the tools that have been used during this project, is the Eclipse Platform. The choice of using this tool and development platform for the Overture project seems to have been a success. Developing the plugins containing the functionality for the editor etc. proved

to be fairly easy. Most of the functionality is already available within the platform. By extending the existing classes and providing some code which handles the specific demands of the Overture Development Tools, a working tool set could be created within a limited period of time.
Next to that, the current implementation does not yet employ all of the advanced features of Eclipse. For instance, it is possible to automate the building process (i.e. doing the syntactic and semantic analysis and creating the syntax tree) and even make it a continuous background process. So there is room for improvement.
During this project only one minor point caused some frustration: the lack of proper documentation. Although the Eclipse Platform can do a lot, little of it is documented or demonstrated in tutorials. However, this can be overcome by studying the implementation of, for instance, the Java Development Tools.

## 6.5 Experiences with the chosen development approach

The project was carried out in an agile-processing style, meaning that the concepts and implementation (prototype) were simultaneously worked at. After each period in which a concept was prototyped (i.e. the compiler, the XML schema and the XML processor and the code for the Eclipse environment), a part of the thesis was written concerning that subject.
Throughout the whole duration of the project, meetings took place with the supervisors once every two months.

This way of working seems to have worked quite well. Especially writing a chapter of the thesis after a part of the Overture Development Tools had been created, helped to focus the attention around some of the difficult points of that part. This helped to properly motivate the chosen solution or re-evaluate that solution and modify it in order to come up with something better.
This was especially useful during the adaption of JavaCC. The design of this tool made it very difficult to enable the generated compiler to recover from syntax errors. Looking back at the options that were available and writing down how these could be used and combined, helped to come up with a working solution.

# Chapter 7

# Future work

The current state of the Overture Development Tools does not yet permit it to be used for real projects. Especially the lack of a static semantics checker limits the usefulness of the ODT. The implementation of this part of the tool set should probably be done using the work of Marcel Verhoef [Ver92], who has written a static semantics checker for the IPTES Meta-IV language.
Furthermore, it is possible to use some of the more advanced concepts of Eclipse. An example of this is the use of so-called incremental project builders and project natures. Incremental project builders provide a mechanism for processing resources in a project and producing some build output. The builder framework makes it easier to incrementally maintain that build state as the input resources change. Project natures, in turn, create a concrete connection between a tool and a project in the Eclipse workspace, and they provide support for persisting, sharing, and managing the relationship of the tool with the project. Natures and builders are often used together, as this provides a way to install and remove builders at appropriate times, and allows for builder ordering based on nature dependencies [Art03].
Other functionality, that could be added to the Overture Development Tools, might be an interpreter and maybe even some debugging functionality. This would allow the user to do checks beyond those that can be done during compile time. The syntax and static semantics checker can find a lot of errors, but a mistake in the design of the specification will probably not be found by these two tools. A debugger would make this possible.
Finally, careful thought has to be given on how a user of the tool set can be aided in the design of his application. For example, often it is helpful when it is possible to visualize the design of an application (e.g. using UML). An

UML representation of the specification can relatively easily be generated from the XML that is already available.

On the long term, a lot of functionality can be created to help a user to create his application using the Overture Modeling Language. Doing so is an interesting task which will need more research and attention.

# Bibliography

[Aho86] Aho, Alfred V., Ravi Sethi, Jeffrey D. Ullman, 1986, *Compilers: Principles , techniques and tools*, Addison Wesley.

[Art03] Arthorne, John, 2003, *Project Builders and Natures*. Available at: `http://www.eclipse.org/articles/Article-Builders/builders.html` (Accessed: 17-06-2004)

[Bin03] Binstock, Cliff, Dave Peterson, Mitchell Smith, Mike Wooding, Chris Dix, Chris Galtenberg, 2003, *The XML Schema complete reference*, Addison Wesley.

[Bur83] Burke, M.G., 1983, *A practical method for LR and LL syntactic error diagnosis and recovery*, PhD thesis, Department of Computer Science, New York University.

[Cos00] Costello, Roger L., 2000, *Best practices: XML Schema*. Available at: `http://www.xfront.com/xml2000/index.htm` (Visited: 18-05-2004)

[DOM] DOM Specification. Available at: `http://www.w3.org/TR/REC-DOM-Level-2-Core/` (Visited: 19-05-2004)

[Fit04] Fitzgerald, John, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, Marcel Verhoef, 2004, *Validated designs for object-oriented systems*, Springer.

[Gam96] Gamma, Erich, Richard Helm, Ralph Johnson, John Vlissides, 1996, *Design patterns: Elements of reusable object-oriented software*, 6th edition, Addison-Wesley professional computing series, pp.331-349.

[Gul02] Gulbransen, David, 2002, *Special Edition: Using XML Schema*, QUE.

[Gru00] Grune, D., J. Bal, C. Jacobs, K. Langendoen, 2000, *Modern compiler design*, 1st edition, John Wiley & Sons.

[Har03] Harold, Elliotte Rusty, 2003, *Processing XML with Java*, Addison Wesley.

[Hol95] Holmes, Jim, 1995, *Object-oriented compiler construction*, Prentice Hall Inc.

[Idr99] Idris, Nazmul, 1999, *Should I use SAX or DOM?*.
Available at: `http://developerlife.com/saxvsdom/default.htm` (Visited: 19-05-2004)

[JGF] JavaCC [tm]: Grammar Files.
Available at: `https://javacc.dev.java.net/doc/javaccgrm.html` (Visited: 27-04-2004)

[JTB] Java Tree Builder documentation.
Available at: `http://www.cs.purdue.edu/jtb/docs.html` (Visited: 27-04-2004)

[Pla88] Plat, Nico, 1988, *Towards a VDM-SL compiler*, Master's thesis, Department of Computer Science, Delft University of Technology.

[SAX] SAX Spefication.
Available at: `http://www.saxproject.org/` (Visited: 19-05-2004)

[Sha03] Shavor, Sherry, Jim D'Anjou, Scott Fairbrother, Dan Kehn, John Kellerman, Pat McCarthy, 2003, *The Java developer's guide to Eclipse*, Addison-Wesley.

[Spe04] Spek, Pieter van der, 2004, *The Overture Project: Towards an open source tool set*, Research thesis, Department of Computer Science, Delft University of Technology.

[Ver92] Verhoef, Marcel, 1992, *A constructive semantics for the IPTES Meta-IV specification language*, Master's Thesis, Department of Computer Science, Delft University of Technology.

[Vli02] Vlist, Eric van der, 2002, *XML Schema*, O'Reilly.

[Wal02] Walmsley, Priscilla, 2002, *Definitive: XML Schema*, Prentice Hall PTR.

[Wil95] Wilhelm, Reinhard, Dieter Maurer, 1995, *Compiler design*, Addison-Wesley.

# Appendix A

# The Overture Language definition

## A.1 Changes

The following productions have changed:

1. Type, value, function, operation and instance variable definitions
   The original definitions where:

```
type definitions = 'types',
    [ access type definition ,
      { ';', access type definition },
      [ ';' ]
    ] ;
value definitions = 'values',
    [ access value definition ,
      { ';', access value definition },
      [ ';' ]
    ] ;
function definitions = 'functions',
    [ access function definition ,
      { ';', access function definition },
      [ ';' ]
```

```
    ] ;
operation definitions = 'operations',
    [ access operation definition,
      { ';', access operation definition },
      [ ';' ]
    ] ;
instance variable definitions = 'instance', 'variables',
    [ instance variable definition,
      { ';', instance variable definition }
    ] ;
```

The changed productions are shown below:

```
type definitions = 'types',
     [ access type definition , ';',
       { access type definition, ';' }
     ] ;
value definitions = 'values',
     [ access value definition, ';',
       { access value definition, ';' }
     ] ;
function definitions = 'functions',
     [ access function definition, ';',
       { access function definition, ';' }
     ] ;
operation definitions = 'operations',
     [ access operation definition, ';',
       { access operation definition, ';' }
     ] ;
instance variable definitions = 'instance', 'variables',
     [ instance variable definition, ';',
       { instance variable definitionm, ';' }
     ] ;
```

2. Permission predicates
   The original definition was:

```
permission predicates = permission predicate,
     { ';', permission predicate } ;
```

The changed production is shown below:

```
permission predicates = permission predicate , ';' ,
     { permission predicate , ';' } ;
```

3. Tuple constructor
   The original definition was:

```
tuple constructor = 'mk_', '(', expression ,
     expression list , ')' ;
```

   The changed production is shown below:

```
tuple constructor = 'mk_', '(', expression ,
     ',' , expression list , ')' ;
```

4. Same class, same base class, base class and class membership expressions
   The original productions where:

```
isofbaseclass expression = 'isofbaseclass',
     '(', name, expression , ')' ;
isofclass expression = 'isofclass',
     '(', name, expression , ')' ;
samebaseclass expression = 'samebaseclass',
     '(', expression , expression , ')' ;
sameclass expression = 'sameclass',
     '(', expression , expression , ')' ;
```

   The changed productions are:

```
isofbaseclass expression = 'isofbaseclass',
     '(', name, ',' , expression , ')' ;
isofclass expression = 'isofclass',
     '(', name, ',' , expression , ')' ;
samebaseclass expression = 'samebaseclass',
     '(', expression , ',' , expression , ')' ;
```

```
sameclass expression = 'sameclass',
     '(', expression , ',' , expression , ')' ;
```

5. Block statement
   The original production was:

```
block statement = '(', { dcl statement },
     statement , { ';', statement }, [ ';' ], ')' ;
```

   The changed production is:

```
block statement = '(', { dcl statement },
     statement , ';' , { statement , ';' }, ')' ;
```

## A.2 Language definition

This appendix specifies the complete syntax for the Overture Modeling Language.

### A.2.1 Document

```
document = class , { class } ;
```

### A.2.2 Classes

```
class = 'class', identifier ,
    [ inheritance clause ],
    [ class body ],
    'end', identifier ;
```

```
inheritance clause = 'is subclass of',
    identifier, { identifier } ;
```

### A.2.3 Definitions

```
class body = definition block, { definition block } ;
definition block = type definitions
| value definitions
| function definitions
| operation definitions
| instance variable definitions
| synchronization definitions
| thread definitions ;
```

#### A.2.3.1 Type Definitions

```
type definitions = 'types',
    [ access type definition , ';' ,
      { access type definition , ';' } ] ;
access type definition =
       (['access'] , ['static'])
     | (['static'] , ['access']) ,
    type definition ;
access = 'public'
| 'private'
| 'protected' ;
type definition = identifier, '=', type, [ invariant ]
| identifier, '::', field list, [ invariant ] ;
type = bracketed type
| basic type
| composite type
| union type
| product type
| optional type
| set type
| seq type
```

```
    | map type
    | partial function type
    | type name
    | type variable ;
    bracketed type = '(', type, ')' ;
    basic type = 'bool' | 'nat' | 'nat1' | 'int'
        | 'rat' | 'real' | 'char' | 'token' ;
    composite type = 'compose', identifier,
        'of', field list, 'end' ;
    field list = { field } ;
    field = [ identifier, ':' ], type
    | [ identifier, ':-' ], type ;
    union type = type, '|', type, { '|', type } ;
    product type = type, '*', type, { '*', type } ;
    optional type = '[', type, ']' ;
    set type = 'set of', type ;
    seq type = seq0 type
    | seq1 type ;
    seq0 type = 'seq of', type ;
    seq1 type = 'seq1 of', type ;
    map type = general map type
    | injective map type ;
    general map type = 'map', type, 'to', type ;
    injective map type = 'inmap', type, 'to', type ;
    function type = partial function type ;
    partial function type = discretionary type, '->', type ;
    discretionary type = type
    | '(', ')' ;
    type name = name ;
    type variable = type variable identifier ;
    invariant = 'inv', invariant initial function ;
    invariant initial function = pattern, '==', expression ;
```

#### A.2.3.2 Value Definitions

```
    value definitions = 'values',
        [ access value definition, ';',
          { access value definition , ';'} ] ;
    access value definition =
           (['access'] , ['static'])
```

```
        | (['static'] , ['access']) ,
      value definition;
  value definition = pattern , [ ':', type ], '=', expression ;
```

#### A.2.3.3 Function Definitions

```
  function definitions = 'functions',
      [ access function definition , ';',
        { access function definition , ';' } ] ;
  access function definition =
         (['access'] , ['static'])
       | (['static'] , ['access']) ,
      function definition ;
  function definition = explicit function definition
  | extended explicit function definition ;
  explicit function definition = identifier ,
      [ type variable list ], ':', function type,
      identifier , parameters list , '==', function body,
      [ 'pre', expression ] , [ 'post', expression ] ;
  extended explicit function definition = identifier ,
      [ type variable list ], parameter types,
      identifier type pair list , '==', function body,
      [ 'pre', expression ], [ 'post', expression ] ;
  type variable list = '[', type variable identifier ,
      { ',', type variable identifier }, ']' ;
  identifier type pair = identifier , ':', type ;
  parameter types = '(', [ pattern type pair list ], ')' ;
  identifier type pair list = identifier , ':', type,
      { ',', identifier , ':', type } ;
  pattern type pair list = pattern list , ':', type,
      { ',', pattern list , ':', type } ;
  parameters list = parameters, { parameters } ;
  parameters = '(', [ pattern list ], ')' ;
  function body = expression
  | 'is not yet specified'
  | 'is subclass responsibility' ;
```

#### A.2.3.4 Operation Definitions

```
operation definitions = 'operations',
    [ access operation definition, ';',
      { access operation definition, ';' } ] ;
access operation definition =
       (['access'] , ['static'])
     | (['static'] , ['access']) ,
    operation definition ;
operation definition = explicit operation definition
| extended explicit operation definition ;
explicit operation definition = identifier, ':',
    operation type, identifier, parameters, '==',
    operation body, [ 'pre', expression ],
    [ 'post', expression ] ;
extended explicit operation definition = identifier,
    parameter types, [ identifier type pair list ],
    '==', operation body, [ externals ],
    [ 'pre', expression ], [ 'post', expression ],
    [ exceptions ] ;
operation type = discretionary type,
    '==>', discretionary type ;
operation body = statement
| 'is not yet specified'
| 'is subclass responsibility' ;
externals = 'ext', var information, { var information } ;
var information = mode, name list, [ ':', type ] ;
mode = 'rd' | 'wr' ;
exceptions = 'errs', error list ;
error list = error, { error } ;
error = identifier, ':', expression, '->', expression ;
```

#### A.2.3.5 Instance Variable Definitions

```
instance variable definitions = 'instance', 'variables',
    [ instance variable definition, ';',
    { instance variable definition, ';' } ] ;
instance variable definition =
```

```
  access assignment definition
| invariant definition ;
access assignment definition =
       (['access'] , ['static'])
     | (['static'] , ['access']) ,
    assignment definition ;
invariant definition = 'inv', expression ;
init statement = 'init', statement ;
```

#### A.2.3.6 Synchronization Definitions

```
synchronization definitions = 'sync', [ synchronization ] ;
synchronization = permission predicates ;
permission predicates = permission predicate, ';',
    { permission predicate, ';' } ;
permission predicate = 'per', name, '=>', expression
| mutex predicate ;
mutex predicate = 'mutex', '(', 'all', ')'
| 'mutex', '(', name list ')' ;
```

#### A.2.3.7 Thread Definitions

```
thread definitions = 'thread', [ thread definition ] ;
thread definition = periodic thread definition
| procedural thread definition ;
periodic thread definition = periodic obligation ;
periodic obligation = 'periodic','(', numeral, ')',
    '(', name, ')' ;
procedural thread definition = statement ;
```

### A.2.4 Expressions

```
expression list = expression , { ',' , expression } ;
expression = bracketed expression
| let expression
| let be expression
| if expression
| cases expression
| unary expression
| binary expression
| quantified expression
| iota expression
| set enumeration
| set comprehension
| set range expression
| sequence enumeration
| sequence comprehension
| subsequence
| map enumeration
| map comprehension
| tuple constructor
| record constructor
| record modifier
| apply
| field select
| tuple select
| function type instantiation
| lambda expression
| new expression
| self expression
| threadid expression
| general is expression
| undefined expression
| isofbaseclass expression
| isofclass expression
| samebaseclass expression
| sameclass expression
| act expression
| fin expression
| active expression
| req expression
| waiting expression
| name
| old name
```

```
    | symbolic literal ;
```

#### A.2.4.1 Bracketed Expressions

```
    bracketed expression = '(', expression , ')' ;
```

#### A.2.4.2 Local Binding Expressions

```
    let expression = 'let', local definition ,
        { ',', local definition }, 'in', expression ;
    let be expression = 'let', bind ,
        [ 'be', 'st', expression ], 'in', expression ;
```

#### A.2.4.3 Conditional Expressions

```
    if expression = 'if', expression , 'then', expression ,
        { elseif expression }, 'else', expression ;
    elseif expression = 'elseif', expression , 'then',
       expression ;
    cases expression = 'cases', expression , ':',
        cases expression alternatives ,
        [ ',', others expression ], 'end' ;
    cases expression alternatives =
        cases expression alternative ,
        { ',', cases expression alternative } ;
    cases expression alternative = pattern list ,
        '->', expression ;
    others expression = 'others', '->', expression ;
```

#### A.2.4.4 Unary Expressions

```
unary expression = prefix expression
| map inverse ;
prefix expression = unary operator, expression ;
unary operator = unary plus
| unary minus
| arithmetic abs
| floor
| not
| set cardinality
| finite power set
| distributed set union
| distributed set intersection
| sequence head
| sequence tail
| sequence length
| sequence elements
| sequence indices
| distributed sequence concatenation
| map domain
| map range
| distributed map merge ;
unary plus = '+' ;
unary minus = '-' ;
arithmetic abs = 'abs' ;
floor = 'floor' ;
not = 'not' ;
set cardinality = 'card' ;
finite power set = 'power' ;
distributed set union = 'dunion' ;
distributed set intersection = 'dinter' ;
sequence head = 'hd' ;
sequence tail = 'tl' ;
sequence length = 'len' ;
sequence elements = 'elems' ;
sequence indices = 'inds' ;
distributed sequence concatenation = 'conc' ;
map domain = 'dom' ;
map range = 'rng' ;
distributed map merge = 'merge' ;
map inverse = 'inverse', expression ;
```

#### A.2.4.5 Binary Expressions

```
binary expression = expression ,
    binary operator , expression ;
binary operator = arithmetic plus
| arithmetic minus
| arithmetic multiplication
| arithmetic divide
| arithmetic integer division
| arithmetic rem
| arithmetic mod
| less than
| less than or equal
| greater than
| greater than or equal
| equal
| not equal
| or
| and
| imply
| logical equivalence
| in set
| not in set
| subset
| proper subset
| set union
| set difierence
| set intersection
| sequence concatenate
| map or sequence modify
| map merge
| map domain restrict to
| map domain restrict by
| map range restrict to
| map range restrict by
| iterate ;
arithmetic plus = '+' ;
arithmetic minus = '-' ;
```

```
arithmetic multiplication = '*' ;
arithmetic divide = '/' ;
arithmetic integer division = 'div' ;
arithmetic rem = 'rem' ;
arithmetic mod = 'mod' ;
less than = '<' ;
less than or equal = '<=' ;
greater than = '>' ;
greater than or equal = '>=' ;
equal = '=' ;
not equal = '<>' ;
approx = '~=' ;
or = 'or' ;
and = 'and' ;
imply = '=>' ;
logical equivalence = '<=>' ;
in set = 'in set' ;
not in set = 'not in set' ;
subset = 'subset' ;
proper subset = 'psubset' ;
set union = 'union' ;
set difierence = '\' ;
set intersection = 'inter' ;
sequence concatenate = '^' ;
map or sequence modify = '++' ;
map merge = 'munion' ;
map domain restrict to = '<:' ;
map domain restrict by = '<-:' ;
map range restrict to = ':>' ;
map range restrict by = ':->' ;
iterate = '**' ;
```

#### A.2.4.6 Quantified Expressions

```
quantified expression = all expression
| exists expression
| exists unique expression ;
all expression = 'forall', bind list,
    '&', expression ;
exists expression = 'exists', bind list,
```

```
        '&', expression ;
    exists unique expression = 'exists1', bind,
        '&', expression ;
```

#### A.2.4.7 The Iota Expression

```
    iota expression = 'iota', bind, '&', expression ;
```

#### A.2.4.8 Set Expressions

```
    set enumeration = '{', [ expression list ], '}' ;
    set comprehension = '{', expression, '|', bind list,
        [ '&', expression ], '}' ;
    set range expression = '{', expression, ',', '...', ',',
        expression, '}' ;
```

#### A.2.4.9 Sequence Expressions

```
    sequence enumeration = '[', [ expression list ], ']' ;
    sequence comprehension = '[', expression, '|', set bind,
        [ '&', expression ], ']' ;
    subsequence = expression, '(', expression, ',', '...',
        ',', expression, ')' ;
```

#### A.2.4.10 Map Expressions

```
    map enumeration = '{', maplet, { ',', maplet }, '}'
        | '{', '|->', '}' ;
```

```
maplet = expression , '|->' , expression ;
map comprehension = '{' , maplet , '|' , bind list ,
    [ '&' , expression ] , '}' ;
```

#### A.2.4.11 The Tuple Constructor Expression

```
tuple constructor = 'mk_' , '(' , expression , ',' ,
    expression list , ')' ;
```

#### A.2.4.12 Record Expressions

```
record constructor = 'mk_' , name,
    '(' , [ expression list ] , ')' ;
record modifier = 'mu' , '(' , expression , ',',
    record modification , { ',' , record modification },
    ')' ;
record modification = identifier , '|->' , expression ;
```

#### A.2.4.13 Apply Expressions

```
apply = expression , '(' , [ expression list ] , ')' ;
field select = expression , '.' , identifier ;
tuple select = expression , '.#' , numeral ;
function type instantiation = name, '[' , type,
    { ',' , type } , ']' ;
```

#### A.2.4.14 The New Expression

```
new expression = 'new', name,
    '(', [ expression list ], ')' ;
```

#### A.2.4.15 The Self Expression

```
self expression = 'self' ;
```

#### A.2.4.16 The Threadid Expression

```
threadid expression = 'threadid' ;
```

#### A.2.4.17 The Is Expression

```
general is expression = is expression
| type judgement ;
is expression = 'is_', name, '(', expression , ')'
| is basic type, '(', expression , ')' ;
type judgement = 'is_', '(', expression , ',',
   type, ')' ;
```

#### A.2.4.18 The Undefined Expression

```
undefined expression = 'undefined' ;
```

#### A.2.4.19 The Precondition Expression

```
pre-condition expression = 'pre_', '(', expression,
    { ',', expression }, ')' ;
```

#### A.2.4.20 Base Class Membership

```
isofbaseclass expression = 'isofbaseclass', '(', name,
    ',', expression, ')' ;
```

#### A.2.4.21 Class Membership

```
isofclass expression = 'isofclass', '(', name, ',',
    expression, ')' ;
```

#### A.2.4.22 Same Base Class Membership

```
samebaseclass expression = 'samebaseclass', '(',
    expression, ',', expression, ')' ;
```

#### A.2.4.23 Same Class Membership

```
sameclass expression = 'sameclass', '(', expression,
    ',', expression, ')' ;
```

#### A.2.4.24 History Expressions

```
act expression = '#act', '(', name, ')'
| '#act', '(', name list, ')' ;
fin expression = '#fin', '(', name, ')'
| '#fin', '(', name list, ')' ;
active expression = '#active', '(', name, ')'
| '#active', '(', name list, ')' ;
req expression = '#req', '(', name, ')'
| '#req', '(', name list, ')' ;
waiting expression = '#waiting', '(', name, ')'
| '#waiting', '(', name list, ')' ;
```

#### A.2.4.25 Names

```
name = identifier, [ '`', identifier ] ;
name list = name, { ',', name } ;
old name = identifier, '~' ;
```

### A.2.5 State Designators

```
state designator = name
| field reference
| map or sequence reference ;
field reference = state designator, '.', identifier ;
map or sequence reference = state designator,
    '(', expression, ')' ;
```

### A.2.6 Statements

```
statement = let statement
| let be statement
| block statement
| general assign statement
| if statement
| cases statement
| sequence for loop
| set for loop
| index for loop
| while loop
| nondeterministic statement
| call statement
| specification statement
| start statement
| start list statement
| return statement
| always statement
| trap statement
| recursive trap statement
| exit statement
| error statement
| identity statement ;
```

#### A.2.6.1 Local Binding Statements

```
let statement = 'let', local definition,
    { ',', local definition }, 'in', statement ;
local definition = value definition
| function definition ;
let be statement = 'let', bind, [ 'be', 'st',
    expression ], 'in', statement ;
equals definition = pattern bind, '=', expression ;
```

#### A.2.6.2 Block and Assignment Statements

```
block statement = '(', { dcl statement }, statement,
```

```
        ';', { statement , ';' }, ')' ;
    dcl statement = 'dcl', assignment definition,
        { ',', assignment definition }, ';' ;
    assignment definition = identifier, ':', type,
        [ ':=', expression ] ;
    general assign statement = assign statement
    | multiple assign statement ;
    assign statement = state designator, ':=', expression ;
    multiple assign statement = 'atomic', '(' ,
        assign statement, ';', assign statement,
        { ';', assign statement }, ')' ;
```

#### A.2.6.3 Conditional Statements

```
    if statement = 'if', expression, 'then', statement,
        { elseif statement }, [ 'else', statement ] ;
    elseif statement = 'elseif', expression,
        'then', statement ;
    cases statement = 'cases', expression, ':',
        cases statement alternatives,
        [ ',', others statement ], 'end' ;
    cases statement alternatives =
        cases statement alternative,
        { ',', cases statement alternative } ;
    cases statement alternative = pattern list,
        '->', statement ;
    others statement = 'others', '->', statement ;
```

#### A.2.6.4 Loop Statements

```
    sequence for loop = 'for', pattern bind, 'in',
        [ 'reverse' ], expression, 'do', statement ;
    set for loop = 'for', 'all', pattern, 'in set',
        expression, 'do', statement ;
    index for loop = 'for', identifier, '=', expression,
        'to', expression, [ 'by', expression ],
```

```
        'do', statement ;
    while loop = 'while', expression, 'do', statement ;
```

#### A.2.6.5 The Nondeterministic Statement

```
    nondeterministic statement = '||', '(', statement,
        { ',', statement }, ')' ;
```

#### A.2.6.6 Call and Return Statements

```
    call statement = [ object designator, '.' ], name,
        '(', [ expression list ], ')', ;
    object designator = name
    | self expression
    | new expression
    | object field reference
    | object apply ;
    object field reference = object designator, '.',
        identifier ;
    object apply = object designator,
        '(', [ expression list ], ')' ;
    return statement = 'return', [ expression ] ;
```

#### A.2.6.7 The Specification Statement

```
    specification statement = '[', implicit operation body,
        ']' ;
    implicit operation body = [ externals ],
        [ 'pre', expression ],
        'post', expression, [ exceptions ] ;
```

#### A.2.6.8 Start and Start List Statements

```
start statement = 'start', '(', expression, ')' ;
start list statement = 'startlist', '(', expression,
    ')' ;
```

#### A.2.6.9 Exception Handling Statements

```
always statement = 'always', statement,
    'in', statement ;
trap statement = 'trap', pattern bind,
    'with', statement, 'in', statement ;
recursive trap statement = 'tixe', traps,
    'in', statement ;
traps = '{', pattern bind, '|->', statement,
    { ',', pattern bind, '|->', statement }, '}' ;
exit statement = 'exit', [ expression ] ;
```

#### A.2.6.10 The Error Statement

```
error statement = 'error' ;
```

#### A.2.6.11 The Identity Statement

```
identity statement = 'skip' ;
```

### A.2.7 Patterns and Bindings

#### A.2.7.1 Patterns

```
pattern = pattern identifier
| match value
| set enum pattern
| set union pattern
| seq enum pattern
| seq conc pattern
| tuple pattern
| record pattern ;
pattern identifier = identifier | '-' ;
match value = '(', expression, ')'
| symbolic literal ;
set enum pattern = '{', [ pattern list ], '}' ;
set union pattern = pattern, 'union',
    pattern ;
seq enum pattern = '[', [ pattern list ], ']' ;
seq conc pattern = pattern, '^', pattern ;
tuple pattern = 'mk_', '(', pattern, ',',
    pattern list, ')' ;
record pattern = 'mk_', name, '(',
    [ pattern list ], ')' ;
pattern list = pattern, { ',', pattern } ;
```

#### A.2.7.2 Bindings

```
pattern bind = pattern | bind ;
bind = set bind | type bind ;
set bind = pattern, 'in set', expression ;
type bind = pattern, ':', type ;
bind list = multiple bind,
    { ',', multiple bind } ;
multiple bind = multiple set bind
| multiple type bind ;
multiple set bind = pattern list,
    'in set', expression ;
```

```
multiple type bind = pattern list ,
    ':', type ;
type bind list = type bind ,
    { ',', type bind } ;
```

# Appendix B

# JavaCC grammar for OML

```
PARSER_BEGIN(OvertureCompiler)

package org.overture.compiler;

public class OvertureCompiler {}

PARSER_END(OvertureCompiler)


/*****************************************/
/*    Token definitions have been left out    */
/*****************************************/



/*****************************************/
/* THE OVERTURE LANGUAGE GRAMMAR STARTS HERE */
/*****************************************/

void document() : {}
{
  (klasse())+
  <EOF>
}

void klasse() : {}
```

```
{
  "class" <IDENTIFIER> [inheritance_clause()]
  [class_body()]
  "end" <IDENTIFIER>
}

void inheritance_clause() : {}
{
  "is" "subclass" "of"
  <IDENTIFIER> (more_parents())*
}

void more_parents() : {}
{
  "," <IDENTIFIER>
}

void class_body() : {}
{
  (definition_block())+
}

void definition_block() : {}
{
  type_definitions()
| value_definitions()
| function_definitions()
| operation_definitions()
| instance_variable_definitions()
| synchronization_definitions()
| thread_definitions()
}

void type_definitions() : {}
{
  "types" [types_body()]
}

void types_body() : {}
{
  access_type_definition() ";"
  (access_type_definition_opt())*
}
```

```
void access_type_definition_opt() : {}
{
  access_type_definition() ";"
}

void access_type_definition() : {}
{
  [modifiers()] type_definition()
}

void type_definition() : {}
{
  <IDENTIFIER> type_definition_choice()
  [invariant()]
}

void type_definition_choice() : {}
{
  "=" type()
| "::" field_list()
}

void function_type() : {}
{
  type()
}

void type() : {}
{
  union_type()
  (LOOKAHEAD(2) type_opt())*
}

void type_opt() : {}
{
  ("->" | "+>") union_type()
}

void union_type() : {}
{
  product_type()
  (LOOKAHEAD(2) union_type_opt())*
```

```
  }

  void union_type_opt() : {}
  {
    "|" product_type()
  }

  void product_type() : {}
  {
    map_type()
    (LOOKAHEAD(2) product_type_opt())*
  }

  void product_type_opt() : {}
  {
    "*" map_type()
  }

  void map_type() : {}
  {
    general_map_type()
  | injective_map_type()
  | unary_type()
  }

  void general_map_type() : {}
  {
    "map" map_type() "to" map_type()
  }

  void injective_map_type() : {}
  {
    "inmap" map_type() "to" map_type()
  }

  void unary_type() : {}
  {
    seq_or_set_type()
  | primary_type()
  }

  void seq_or_set_type() : {}
  {
```

```
    ("seq" | "seq1" | "set") "of"
    unary_type()
  }

  void primary_type() : {}
  {
    basic_type()
  | quote_type()
  | composite_type()
  | optional_type()
  | type_name()
  | type_variable()
  | bracketed_type()
  }

  void basic_type() : {}
  {
    "nat"
  | "int"
  | "rat"
  | "bool"
  | "nat1"
  | "real"
  | "char"
  | "token"
  }

  void quote_type() : {}
  {
    <QUOTE_LITERAL>
  }

  void composite_type() : {}
  {
    "compose" <IDENTIFIER> "of"
    field_list()
    "end"
  }

  void field_list() : {}
  {
    (field())*
  }
```

```
void field() : {}
{
  [LOOKAHEAD(2) field_id()] type()
}

void field_id() : {}
{
  <IDENTIFIER> (":" | ":-")
}

void type_name() : {}
{
  name()
}

void optional_type() : {}
{
  "[" type() "]"
}

void bracketed_type() : {}
{
  "(" [type()] ")"
}

void type_variable() : {}
{
  <TYPE_VARIABLE_IDENTIFIER>
}

void invariant() : {}
{
  "inv" pattern() "==" expression()
}

void value_definitions() : {}
{
  "values" [values_body()]
}

void values_body() : {}
{
```

```
    access_value_definition() ";"
    (access_value_definition_opt())*
  }

  void access_value_definition_opt() : {}
  {
    access_value_definition() ";"
  }

  void access_value_definition() : {}
  {
    [modifiers()] value_definition()
  }

  void value_definition() : {}
  {
    pattern() [value_definition_type()]
    "=" expression()
  }

  void value_definition_type() : {}
  {
    ":" type()
  }

  void function_definitions() : {}
  {
    "functions" [functions_body()]
  }

  void functions_body() : {}
  {
    access_function_definition() ";"
    (access_function_definition_opt())*
  }

  void access_function_definition_opt() : {}
  {
    access_function_definition() ";"
  }

  void access_function_definition() : {}
  {
```

```
    [modifiers()] function_definition()
  }

  void function_definition() : {}
  {
    <IDENTIFIER> [type_variable_list()]
    (
    explicit_function_definition()
    |
    not_explicit_function_definition()
    )
  }

  void explicit_function_definition() : {}
  {
     ":" function_type() <IDENTIFIER>
     parameters_list() "=="
     function_body() [pre()] [post()]
  }

  void not_explicit_function_definition() : {}
  {
    parameter_types() identifier_type_pair_list()
    extended_explicit_function_definition()
  }

  void extended_explicit_function_definition() : {}
  {
    "==" function_body() [pre()] [post()]
  }

  void pre() : {}
  {
    "pre" expression()
  }

  void post() : {}
  {
    "post" expression()
  }

  void type_variable_list() : {}
  {
```

```
    "[" <TYPE_VARIABLE_IDENTIFIER>
    (type_variable_list_opt())* "]"
  }

  void type_variable_list_opt() : {}
  {
    "," <TYPE_VARIABLE_IDENTIFIER>
  }

  void parameter_types() : {}
  {
    "(" [pattern_type_pair_list()] ")"
  }

  void identifier_type_pair_list() : {}
  {
    identifier_type_pair()
    (identifier_type_pair_list_opt())*
  }

  void identifier_type_pair_list_opt() : {}
  {
    "," identifier_type_pair()
  }

  void identifier_type_pair() : {}
  {
    <IDENTIFIER> ":" type()
  }

  void pattern_type_pair_list() : {}
  {
    pattern_list() ":" type()
    (pat_typepair_list_opt())*
  }

  void pat_typepair_list_opt() : {}
  {
    "," pattern_list() ":" type()
  }

  void parameters_list() : {}
  {
```

```
  (parameters())+
}

void parameters() : {}
{
  "(" [pattern_list()] ")"
}

void function_body() : {}
{
  expression()
| unknown_body()
}

void unknown_body() : {}
{
  "is"
  ( "not" "yet" "specified"
  | "subclass" "responsibility"
  )
}

void operation_definitions() : {}
{
  "operations" [operations_body()]
}

void operations_body() : {}
{
  access_operation_definition() ";"
  (access_operation_definition_opt())*
}

void access_operation_definition_opt() : {}
{
  access_operation_definition() ";"
}

void access_operation_definition() : {}
{
  [modifiers()] operation_definition()
}
```

```
void operation_definition() : {}
{
  <IDENTIFIER>
  (
  explicit_operation_definition()
  |
  not_explicit_operation_definition()
  )
}

void explicit_operation_definition() : {}
{
  ":" operation_type() <IDENTIFIER>
  parameters() "==" operation_body()
  [pre()] [post()]
}

void not_explicit_operation_definition() : {}
{
  parameter_types() [identifier_type_pair_list()]
  extended_explicit_operation_definition()
}

void implicit_operation_body() : {}
{
  [externals()] [pre()] post() [exceptions()]
}

void extended_explicit_operation_definition() : {}
{
  "==" operation_body() [externals()]
  [pre()] [post()] [exceptions()]
}

void operation_type() : {}
{
  type() "==>" type()
}

void operation_body() : {}
{
  statement()
| unknown_body()
```

```
}

void externals() : {}
{
  "ext" (var_information())+
}

void var_information() : {}
{
  mode() name_list() [var_information_type()]
}

void var_information_type() : {}
{
  ":" type()
}

void mode() : {}
{
  "rd"
| "wr"
}

void exceptions() : {}
{
  "errs" (errs_list())+
}

void errs_list() : {}
{
  <IDENTIFIER> ":" expression()
  "->" expression()
}

void instance_variable_definitions() : {}
{
  "instance" "variables"
  [instance_variables_body()]
}

void instance_variables_body() : {}
{
  instance_variable_definition() ";"
```

```
    (instance_variable_definition_opt())*
  }

  void instance_variable_definition_opt() : {}
  {
    instance_variable_definition() ";"
  }

  void instance_variable_definition() : {}
  {
    access_assignment_definition()
  | invariant_definition()
  }

  void access_assignment_definition() : {}
  {
    [modifiers()] assignment_definition()
  }

  void invariant_definition() : {}
  {
    "inv" expression()
  }

  void init_statement() : {}
  {
    "init" statement()
  }

  void synchronization_definitions() : {}
  {
    "sync" [sync_body()]
  }

  void sync_body() : {}
  {
    sync_predicate() ";"
    (sync_predicate_opt())*
  }

  void sync_predicate_opt() : {}
  {
    sync_predicate() ";"
```

```
}

void sync_predicate() : {}
{
  sync_per()
| sync_mutex()
}

void sync_per() : {}
{
  "per" name() "=>" expression()
}

void sync_mutex() : {}
{
  "mutex" "(" ("all" | name_list()) ")"
}

void thread_definitions() : {}
{
  "thread" [thread_body()]
}

void thread_body() : {}
{
  periodic_thread_definition()
| procedural_thread_definition()
}

void periodic_thread_definition() : {}
{
  "periodic"
  "(" <NUMERIC_LITERAL> ")"
  "(" name() ")"
}

void procedural_thread_definition() : {}
{
  statement()
}

void expression_list() : {}
{
```

```
    expression()
    (LOOKAHEAD(2) expression_list_opt())*
  }

  void expression_list_opt() : {}
  {
    "," expression()
  }

  void expression() : {}
  {
    logical_equivalence_expression()
  }

  void primary_expression() : {}
  {
    let_expression()
  | if_expression()
  | cases_expression()
  | quantified_expression()
  | iota_expression()
  | curly_expression()
  | sequence_expression()
  | tuple_constructor()
  | record_constructor()
  | record_modifier()
  | new_expression()
  | general_is_expression()
  | undefined_expression()
  | isofbaseclass_expression()
  | isofclass_expression()
  | samebaseclass_expression()
  | sameclass_expression()
  | act_expression()
  | fin_expression()
  | active_expression()
  | req_expression()
  | waiting_expression()
  | self_expression()
  | threadid_expression()
  | precondition_expression()
  | bracket()
  | name_expression()
```

```
    | SYMBOLIC_LITERAL()
    | result_expression()
    }

    void result_expression() : {}
    {
      "RESULT"
    }

    void expression_tail() : {}
    {
      select()
    | bracket()
    }

    void bracket() : {}
    {
      "("
      [expression_list()]
      ( subsequence()
      | apply_or_bracketed()
      )
    }

    void subsequence() : {}
    {
      "," "..." "," expression() ")"
    }

    void apply_or_bracketed() : {}
    {
      ")"
    }

    void select() : {}
    {
      "."
      ( field_select()
      | tuple_select()
      )
    }

    void field_select() : {}
```

```
  {
    <IDENTIFIER>
  }

  void tuple_select() : {}
  {
    "#" <NUMERIC_LITERAL>
  }

  void logical_equivalence_expression() : {}
  {
    imply_expression()
    (LOOKAHEAD(2) logical_equivalence_opt())*
  }
  void logical_equivalence_opt() : {}
  {
    "<=>" imply_expression()
  }

  void imply_expression() : {}
  {
    or_expression()
    (LOOKAHEAD(2) imply_opt())*
  }
  void imply_opt() : {}
  {
    "=>" or_expression()
  }

  void or_expression() : {}
  {
    and_expression()
    (LOOKAHEAD(2) or_opt())*
  }

  void or_opt() : {}
  {
    "or" and_expression()
  }

  void and_expression() : {}
  {
    not_expression()
```

```
  (LOOKAHEAD(2) and_opt())*
}

void and_opt() : {}
{
  "and" not_expression()
}

void not_expression() : {}
{
  "not" not_expression()
| relational_expression()
}

void relational_expression() : {}
{
  plus_expression()
  (LOOKAHEAD(2) relational_opt())*
}

void relational_opt() : {}
{
  relational_operator() plus_expression()
}

void relational_operator() : {}
{
  ("not" "in" "set"|"in" "set"
  |"psubset"|"subset"|"<>"|"="|">"
  |">="|"<"|"<=")
}

void plus_expression() : {}
{
  star_expression()
  (LOOKAHEAD(2) plus_opt())*
}

void plus_opt() : {}
{
  plus_operator() star_expression()
}
```

```
void plus_operator() : {}
{
  ("+"|"-"|"^"|"++"|"\\"
  |"munion"|"union")
}

void star_expression() : {}
{
  inverse_expression()
  (LOOKAHEAD(2)star_opt())*
}

void star_opt() : {}
{
  star_operator() inverse_expression()
}

void star_operator() : {}
{
  ("*"|"/"|"inter"|"div"|"mod"|"rem")
}

void inverse_expression() : {}
{
  "inverse" inverse_expression()
| map_domain_expression()
}

void map_domain_expression() : {}
{
  map_range_expression()
  (LOOKAHEAD(2) map_domain_opt())*
}

void map_domain_opt() : {}
{
  map_domain_operator()
  map_range_expression()
}

void map_domain_operator() : {}
{
  ("<:"|"<-:")
```

```
}

void map_range_expression() : {}
{
  unary_expression()
  (LOOKAHEAD(2) map_range_opt())*
}

void map_range_opt() : {}
{
  map_range_operator() unary_expression()
}

void map_range_operator() : {}
{
  (":>" | ":->")
}

void unary_expression() : {}
{
  unary_operator() unary_expression()
| comp_expression()
}

void unary_operator() : {}
{
  ("inds" | "conc" | "tl" | "hd" | "elems" | "len"
  | "merge" | "rng" | "dom" | "dunion" | "dinter"
  | "power" | "card" | "floor" | "abs" | "-" | "+")
}

void comp_expression() : {}
{
  iterate_expression()
  (LOOKAHEAD(2) comp_opt())*
}

void comp_opt() : {}
{
  "comp" iterate_expression()
}

void iterate_expression() : {}
```

```
  {
    primary_expression()
    (LOOKAHEAD(2) expression_tail())*
    (LOOKAHEAD(2) iterate_opt())*
  }

  void iterate_opt() : {}
  {
    "**" primary_expression()
    (LOOKAHEAD(2) expression_tail())*
  }

  void let_expression() : {}
  {
    "let"
    ( LOOKAHEAD(bind() "be" "st") let_be_expression()
    | LOOKAHEAD(bind() "in") let_be_expression()
    | let_normal_expression()
    )
  }

  void let_normal_expression() : {}
  {
    local_definition()
    (let_expression_opt())* "in" expression()
  }

  void let_expression_opt() : {}
  {
    "," local_definition()
  }

  void let_be_expression() : {}
  {
    bind() [be_st_expression()] "in" expression()
  }

  void be_st_expression() : {}
  {
    "be" "st" expression()
  }

  void if_expression() : {}
```

```
{
  "if" expression() "then" expression()
  (elseif_expression())* "else" expression()
}

void elseif_expression() : {}
{
  "elseif" expression() "then" expression()
}

void cases_expression() : {}
{
  "cases" expression() ":"
  cases_alternatives_expression()
  [others_expression()] "end"
}

void cases_alternatives_expression() : {}
{
  pattern_list() "->" expression() (LOOKAHEAD(2)
  cases_alternatives_expression_opt())*
}

void cases_alternatives_expression_opt() : {}
{
  "," pattern_list() "->" expression()
}

void others_expression() : {}
{
  "," "others" "->" expression()
}

void quantified_expression() : {}
{
  ("forall" | "exists" | "exists1")
  bind_list() "&" expression()
}

void iota_expression () : {}
{
  "iota" bind() "&" expression()
}
```

```
void curly_expression() : {}
{
  "{"
  ( LOOKAHEAD( maplet() "|")
    map_comprehension()
  | LOOKAHEAD( maplet())
    map_enumeration()
  | LOOKAHEAD( expression() "|")
    set_comprehension()
  | LOOKAHEAD( expression() "," "...")
    set_range_expression()
  | set_enumeration()
  | "|->" "}"
  )
}

void map_comprehension() : {}
{
  maplet() "|" bind_list()
  [map_comprehension_opt()] "}"
}

void map_comprehension_opt() : {}
{
  "&" expression()
}

void map_enumeration() : {}
{
  maplet() (map_enumeration_opt())* "}"
}

void map_enumeration_opt() : {}
{
  "," maplet()
}

void set_comprehension() : {}
{
  expression() "|" bind_list()
  [set_comprehension_opt()] "}"
}
```

```
void set_comprehension_opt() : {}
{
  "&" expression()
}

void set_range_expression() : {}
{
  expression() "," "..." "," expression() "}"
}

void set_enumeration() : {}
{
  [expression_list()] "}"
}

void sequence_expression() : {}
{
  "[" [expression_list()]
  ( sequence_enumeration()
  | sequence_comprehension()
  )
}

void sequence_enumeration() : {}
{
  "]"
}

void sequence_comprehension() : {}
{
  "|" pattern() set_bind()
  [sequence_comprehension_opt()] "]"
}
void sequence_comprehension_opt() : {}
{
  "&" expression()
}

void maplet() : {}
{
  expression() "|->" expression()
}
```

```
void tuple_constructor() : {}
{
  "mk_" "(" expression_list() ")"
}

void record_constructor() : {}
{
  <RECORD> "(" [expression_list()] ")"
}

void record_modifier() : {}
{
  "mu" "(" expression() ","
  record_modification()
  [record_modifier_opt()] ")"
}

void record_modifier_opt() : {}
{
  "," record_modification()
}

void record_modification() : {}
{
  <IDENTIFIER> "|->" expression()
}

void new_expression() : {}
{
  "new" name() "(" [expression_list()] ")"
}

void self_expression() : {}
{
  "self"
}

void threadid_expression() : {}
{
  "threadid"
}
```

```
void general_is_expression() : {}
{
  is_basic_type()
| type_judgement()
| is_expression()
}

void type_judgement() : {}
{
  "is_" "(" expression() "," type() ")"
}

void is_expression() : {}
{
  <IS_EXPRESSION> "(" expression() ")"
}

void is_basic_type() : {}
{
  <IS_BASIC_TYPE> "(" expression() ")"
}

void undefined_expression() : {}
{
  "undefined"
}

void precondition_expression() : {}
{
  "pre_" "(" expression_list() ")"
}

void isofbaseclass_expression() : {}
{
  "isofbaseclass" "(" name() "," expression() ")"
}

void isofclass_expression() : {}
{
  "isofclass" "(" name() "," expression() ")"
}

void samebaseclass_expression() : {}
```

```
{
  "samebaseclass"
  "(" expression() "," expression() ")"
}

void sameclass_expression() : {}
{
  "sameclass"
  "(" expression() "," expression() ")"
}

void act_expression() : {}
{
  "#act"
  "(" name() (act_expression_opt())* ")"
}

void act_expression_opt() : {}
{
  "," name()
}

void fin_expression() : {}
{
  "#fin"
  "(" name() (fin_expression_opt())* ")"
}

void fin_expression_opt() : {}
{
  "," name()
}

void active_expression() : {}
{
  "#active"
  "(" name() (active_expression_opt())* ")"
}

void active_expression_opt() : {}
{
  "," name()
}
```

```
void req_expression() : {}
{
  "#req"
  "(" name() (req_expression_opt())* ")"
}

void req_expression_opt() : {}
{
  "," name()
}

void waiting_expression() : {}
{
  "#waiting"
  "(" name() (wait_expression_opt())* ")"
}

void wait_expression_opt() : {}
{
  "," name()
}

void name_expression() : {}
{
  <IDENTIFIER>
  [LOOKAHEAD(2) name_expression_opt()]
}

void name_expression_opt() : {}
{
  "~"
| name_expression_tail()
| function_type_instantiation_expression()
}

void name_expression_tail() : {}
{
  "`" <IDENTIFIER>
  [ LOOKAHEAD(2)
    function_type_instantiation_expression()]
}
```

```
void function_type_instantiation_expression() : {}
{
  "[" type()
  (function_type_instantiation_expression_opt())*
  "]"
}

void function_type_instantiation_expression_opt() : {}
{
  "," type()
}

void name() : {}
{
  <IDENTIFIER> ["'" <IDENTIFIER>]
}

void name_list() : {}
{
  name() (name_list_opt())*
}

void name_list_opt() : {}
{
  "," name()
}

void old_name() : {}
{
  <IDENTIFIER> "~"
}

void state_designator() : {}
{
  name() (state_designator_opt())*
}

void state_designator_opt() : {}
{
  "." <IDENTIFIER>
| "(" expression() ")"
}
```

```
void statement() : {}
{
  let_statement()
| block_statement()
| if_statement()
| cases_statement()
| for_loop()
| while_loop()
| nondeterministic_statement()
| specification_statement()
| start_statement()
| start_list_statement()
| return_statement()
| always_statement()
| trap_statement()
| recursive_trap_statement()
| exit_statement()
| error_statement()
| identity_statement()
| call_or_assign_statement()
}

void let_statement() : {}
{
  "let"
  ( LOOKAHEAD(bind() "be" "st") let_be_statement()
  | LOOKAHEAD(bind() "in") let_be_statement()
  | let_normal_statement()
  )
}

void let_normal_statement() : {}
{
  local_definition()
  (LOOKAHEAD(2) let_statement_opt())*
  "in" statement()
}

void let_statement_opt() : {}
{
  "," local_definition()
}
```

```
void let_be_statement() : {}
{
  bind() [be_st_expression()] "in" statement()
}

void local_definition() : {}
{
  LOOKAHEAD(<IDENTIFIER> "(")
    function_definition()
| LOOKAHEAD(<IDENTIFIER> "[")
    function_definition()
| LOOKAHEAD(<IDENTIFIER> "=")
    value_definition()
| LOOKAHEAD(<IDENTIFIER> ":" function_type()
  <IDENTIFIER>)
    function_definition()
| value_definition()
}

void equals_definition() : {}
{
  pattern_bind() "=" expression()
}

void block_statement() : {}
{
 "(" (dcl_statement())* statement() ";"
 (block_statement_opt())* ")"
}

void block_statement_opt() : {}
{
  statement() ";"
}

void dcl_statement() : {}
{
  "dcl" assignment_definition()
  (dcl_statement_opt())* ";"
}

void dcl_statement_opt() : {}
{
```

```
  "," assignment_definition()
}

void assignment_definition() : {}
{
  <IDENTIFIER> ":" type() [ass_def_opt()]
}

void ass_def_opt() : {}
{
  ":=" expression()
}

void call_or_assign_statement() : {}
{
  call_or_assign()
| multiple_assign_statement()
}

void call_or_assign() : {}
{
  designator() [assign()]
}

void designator() : {}
{
  (name() | self_expression() | new_expression())
  (LOOKAHEAD(3) designator_tail())*
}

void designator_tail() : {}
{
  field_reference()
| call_or_apply_or_map_or_sequence()
}

void field_reference() : {}
{
  "." name()
}

void call_or_apply_or_map_or_sequence() : {}
{
```

```
    "(" [expression_list()] ")"
  }

  void assign() : {}
  {
    ":=" expression()
  }

  void multiple_assign_statement() : {}
  {
    "atomic" "(" assign_statement() ";"
    (multiple_assign_statement_opt())+ ")"
  }

  void multiple_assign_statement_opt() : {}
  {
    assign_statement() ";"
  }

  void assign_statement() : {}
  {
    state_designator() ":=" expression()
  }

  void if_statement() : {}
  {
    "if" expression() "then" statement()
    (LOOKAHEAD(2) elseif_stat())*
    [LOOKAHEAD(2) else_stat()]
  }

  void elseif_stat() : {}
  {
    "elseif" expression() "then" statement()
  }

  void else_stat() : {}
  {
    "else" statement()
  }

  void cases_statement() : {}
  {
```

```
  "cases" expression() ":"
  cases_alternatives_statement() [
  LOOKAHEAD(2) others_statement()]
}

void cases_alternatives_statement() : {}
{
  pattern_list() "->" statement()
  (LOOKAHEAD(2) cases_alternatives_statement_opt())*
}

void cases_alternatives_statement_opt() : {}
{
  "," pattern_list() "->" statement()
}

void others_statement() : {}
{
  "," "others" "->" statement()
}

void for_loop() : {}
{
  "for"
  ( LOOKAHEAD(2) sequence_for_loop()
  | set_for_loop()
  | index_for_loop()
  )
}

void sequence_for_loop() : {}
{
  pattern_bind() "in" ["reverse"]
  expression() "do" statement()
}

void set_for_loop() : {}
{
  "all" pattern() "in" "set"
  expression() "do" statement()
}

void index_for_loop() : {}
```

```
{
  <IDENTIFIER> "=" expression() "to"
  expression() [by_expression()] "do"
  statement()
}

void by_expression() : {}
{
  "by" expression()
}

void while_loop() : {}
{
  "while" expression() "do" statement()
}

void nondeterministic_statement() : {}
{
  "||" "(" statement()
  (nondeterministic_statement_opt())* ")"
}

void nondeterministic_statement_opt() : {}
{
  "," statement()
}

void object_designator() : {}
{
  (name() | self_expression() | new_expression())
  (LOOKAHEAD(2) object_designator_opt())*
}

void object_designator_opt() : {}
{
  object_field_reference()
| object_apply()
}

void object_field_reference() : {}
{
  "." <IDENTIFIER>
}
```

```
void object_apply() : {}
{
  "(" [expression_list()] ")"
}

void return_statement() : {}
{
  "return" [expression()]
}

void specification_statement() : {}
{
  "[" implicit_operation_body() "]"
}

void start_statement() : {}
{
  "start" "(" expression() ")"
}

void start_list_statement() : {}
{
  "startlist" "(" expression() ")"
}

void always_statement() : {}
{
  "always" statement() "in" statement()
}

void trap_statement() : {}
{
  "trap" pattern_bind() "with"
  statement() "in" statement()
}

void recursive_trap_statement() : {}
{
  "tixe" traps() "in" statement()
}

void traps() : {}
{
```

```
    "{" pattern_bind() "|->" statement()
    (traps_opt())* "}"
  }

  void traps_opt() : {}
  {
    "," pattern_bind() "|->" statement()
  }

  void exit_statement() : {}
  {
    "exit" [expression()]
  }

  void error_statement() : {}
  {
    "error"
  }

  void identity_statement() : {}
  {
    "skip"
  }

  void pattern() : {}
  {
    (pattern_identifier()
    | match_value()
    | set_enum_pattern()
    | seq_enum_pattern()
    | tuple_pattern()
    | record_pattern())
    (LOOKAHEAD(2) pattern_tail())*
  }

  void pattern_tail() : {}
  {
    ("union" | "^") pattern()
  }

  void pattern_identifier() : {}
  {
    <IDENTIFIER>
```

```
  | "_"
  }

  void match_value() : {}
  {
    "(" expression() ")"
  | SYMBOLIC_LITERAL()
  }

  void set_enum_pattern() : {}
  {
    "{" [pattern_list()] "}"
  }

  void seq_enum_pattern() : {}
  {
    "[" [pattern_list()] "]"
  }

  void tuple_pattern() : {}
  {
    "mk_" "(" pattern() "," pattern_list() ")"
  }

  void record_pattern() : {}
  {
    <RECORD> "(" [pattern_list()] ")"
  }

  void pattern_list() : {}
  {
    pattern() (pattern_list_opt())*
  }

  void pattern_list_opt() : {}
  {
    "," pattern()
  }

  void pattern_bind() : {}
  {
    pattern()
    [ LOOKAHEAD(2) pattern_bind_opt() ]
```

```
}

void pattern_bind_opt() : {}
{
  set_bind()
| type_bind()
}

void bind() : {}
{
  pattern()
  ( set_bind()
  | type_bind()
  )
}

void set_bind() : {}
{
  "in" "set" expression()
}

void type_bind() : {}
{
  ":" type()
}

void bind_list() : {}
{
  multiple_bind() (bind_list_opt())*
}

void bind_list_opt() : {}
{
  "," multiple_bind()
}

void multiple_bind() : {}
{
  pattern_list()
  ( multiple_set_bind()
  | multiple_type_bind()
  )
}
```

```
void multiple_set_bind() : {}
{
  "in" "set" expression()
}

void multiple_type_bind() : {}
{
  ":" type()
}

void type_bind_list() : {}
{
  pattern() type_bind()
  (type_bind_list_opt())*
}

void type_bind_list_opt() : {}
{
  "," pattern() type_bind()
}
```

# Appendix C

# XML Schema definition for the Overture Modeling Language

Documentation for the Schema can also be found online at `http://www.overturetool.org/schema`.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsd:schema
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!--
================================================================
GLOBAL DEFINITIONS
================================================================
-->
  <xsd:complexType name="extra">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="comment"
                   type="xsd:string" />
      <xsd:element minOccurs="0"
                   name="location">
        <xsd:complexType>
          <xsd:attribute name="beginline"
                         type="xsd:int" />
          <xsd:attribute name="endline"
                         type="xsd:int" />
          <xsd:attribute name="begincolumn"
```

```
                        type="xsd:int"/>
          <xsd:attribute name="endcolumn"
                        type="xsd:int"/>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="identifier">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:element name="value" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:attribute name="access">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="public"/>
        <xsd:enumeration value="private"/>
        <xsd:enumeration value="protected"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:attribute>
  <xsd:attribute fixed="static"
                 name="static"
                 type="xsd:string"/>
  <!--
================================================================
LITERAL
================================================================
-->
  <xsd:group name="symbolicliteral">
    <xsd:choice>
      <xsd:element name="boollit" type="boollit"/>
      <xsd:element name="nillit" type="nillit"/>
      <xsd:element name="numlit" type="numlit"/>
      <xsd:element name="reallit" type="reallit"/>
      <xsd:element name="charlit" type="charlit"/>
      <xsd:element name="textlit" type="textlit"/>
      <xsd:element name="quotelit" type="quotelit"/>
    </xsd:choice>
  </xsd:group>
```

```
<xsd:complexType name="boollit">
  <xsd:sequence>
    <xsd:element minOccurs="0"
                 name="extra_information"
                 type="extra"/>
    <xsd:element name="value" type="xsd:boolean"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="nillit">
  <xsd:sequence>
    <xsd:element minOccurs="0"
                 name="extra_information"
                 type="extra"/>
    <xsd:element fixed="nil"
                 name="value"
                 type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="numlit">
  <xsd:sequence>
    <xsd:element minOccurs="0"
                 name="extra_information"
                 type="extra"/>
    <xsd:element name="value" type="xsd:integer"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="reallit">
  <xsd:sequence>
    <xsd:element minOccurs="0"
                 name="extra_information"
                 type="extra"/>
    <xsd:element name="value" type="xsd:float"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="charlit">
  <xsd:sequence>
    <xsd:element minOccurs="0"
                 name="extra_information"
                 type="extra"/>
    <xsd:element name="value" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="textlit">
```

```
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra" />
      <xsd:element name="value" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="quotelit">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra" />
      <xsd:element name="value" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
  <!--
=============================================================
DOCUMENT
=============================================================
-->
  <xsd:element name="project">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element maxOccurs="unbounded"
                     ref="document" />
      </xsd:sequence>
      <xsd:attribute name="projectname"
                     type="xsd:string"
                     use="optional" />
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="document">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element minOccurs="0"
                     name="extra_information"
                     type="extra" />
        <xsd:element maxOccurs="unbounded"
                     ref="class" />
      </xsd:sequence>
      <xsd:attribute name="filename"
                     type="xsd:string"
                     use="required" />
```

```
      <xsd:attribute name="timestamp"
                     type="xsd:dateTime"
                     use="optional"/>
    </xsd:complexType>
  </xsd:element>
  <!--
=============================================================
CLASSES
=============================================================
-->
  <xsd:element name="class">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element minOccurs="0"
                     name="extra_information"
                     type="extra"/>
        <xsd:element name="identifier"
                     type="identifier"/>
        <xsd:element minOccurs="0" name="supercls">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element maxOccurs="unbounded"
                           name="identifier"
                           type="identifier"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
        <xsd:group maxOccurs="unbounded"
                   minOccurs="0"
                   ref="definitions"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <!--
=============================================================
DEFINITIONS
=============================================================
-->
  <xsd:group name="definitions">
    <xsd:choice>
      <xsd:element ref="typedefinition"/>
      <xsd:element ref="valuedefinition"/>
      <xsd:group ref="functiondefinition"/>
```

```
      <xsd:group ref="operationdefinition"/>
      <xsd:group ref="instancevariabledefinition"/>
      <xsd:group ref="syncdefinition"/>
      <xsd:group ref="threaddefinition"/>
    </xsd:choice>
  </xsd:group>
  <!--
================================================================
TYPE DEFINITIONS
================================================================
-->
  <xsd:element name="typedefinition">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element minOccurs="0"
                     name="extra_information"
                     type="extra"/>
        <xsd:element name="identifier"
                     type="identifier"/>
        <xsd:choice>
          <xsd:element maxOccurs="unbounded"
                       name="field"
                       type="field"/>
          <xsd:group ref="type"/>
        </xsd:choice>
        <xsd:element minOccurs="0" ref="invariant"/>
      </xsd:sequence>
      <xsd:attribute ref="static" use="optional"/>
      <xsd:attribute ref="access" use="optional"/>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="invariant">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element minOccurs="0"
                     name="extra_information"
                     type="extra"/>
        <xsd:group ref="pattern"/>
        <xsd:group ref="expression"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:group name="type">
```

```
    <xsd:choice>
      <xsd:element name="basictype"
                   type="basictype"/>
      <xsd:element name="quotetype"
                   type="quotelit"/>
      <xsd:element name="compositetype"
                   type="compositetype"/>
      <xsd:element name="uniontype"
                   type="uniontype"/>
      <xsd:element name="producttype"
                   type="producttype"/>
      <xsd:element name="settype"
                   type="settype"/>
      <xsd:group ref="seqtype"/>
      <xsd:group ref="maptype"/>
      <xsd:element name="typename"
                   type="typename"/>
      <xsd:group ref="functiontype"/>
      <xsd:element name="bracketedtype"
                   type="bracketedtype"/>
      <xsd:element name="optionaltype"
                   type="optionaltype"/>
      <xsd:element name="typevariable"
                   type="typevariable"/>
    </xsd:choice>
  </xsd:group>
  <xsd:complexType name="bracketedtype">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group ref="type"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:simpleType name="basictype">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="bool"/>
      <xsd:enumeration value="nat"/>
      <xsd:enumeration value="nat1"/>
      <xsd:enumeration value="int"/>
      <xsd:enumeration value="rat"/>
      <xsd:enumeration value="real"/>
      <xsd:enumeration value="char"/>
```

```
      <xsd:enumeration value="token"/>
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:complexType name="compositetype">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:element name="identifier"
                   type="identifier"/>
      <xsd:element maxOccurs="unbounded"
                   minOccurs="1"
                   name="field"
                   type="field"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="field">
    <xsd:sequence>
      <xsd:sequence minOccurs="0">
        <xsd:element minOccurs="0"
                     name="extra_information"
                     type="extra"/>
        <xsd:element name="identifier"
                     type="identifier"/>
        <xsd:element name="operator">
          <xsd:complexType>
            <xsd:attribute name="value">
              <xsd:simpleType>
                <xsd:restriction base="xsd:string">
                  <xsd:enumeration value=":"/>
                  <xsd:enumeration value=":-"/>
                </xsd:restriction>
              </xsd:simpleType>
            </xsd:attribute>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
      <xsd:group ref="type"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="uniontype">
    <xsd:sequence>
      <xsd:element minOccurs="0"
```

```
                    name="extra_information"
                    type="extra" />
      <xsd:group maxOccurs="unbounded"
                 minOccurs="2"
                 ref="type" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="producttype">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra" />
      <xsd:group maxOccurs="unbounded"
                 minOccurs="2"
                 ref="type" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="settype">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra" />
      <xsd:group ref="type" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="optionaltype">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra" />
      <xsd:group ref="type" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="typevariable">
    <xsd:sequence>
      <xsd:element name="identifier"
                   type="identifier" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:group name="seqtype">
    <xsd:choice>
      <xsd:element name="seq0type" type="seq0type" />
      <xsd:element name="seq1type" type="seq1type" />
```

```
    </xsd:choice>
  </xsd:group>
  <xsd:complexType name="seq0type">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group ref="type"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="seq1type">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group ref="type"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:group name="maptype">
    <xsd:choice>
      <xsd:element name="generalmaptype"
                   type="generalmaptype"/>
      <xsd:element name="injectivemaptype"
                   type="injectivemaptype"/>
    </xsd:choice>
  </xsd:group>
  <xsd:complexType name="generalmaptype">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group ref="type"/>
      <xsd:group ref="type"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="injectivemaptype">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group ref="type"/>
      <xsd:group ref="type"/>
    </xsd:sequence>
```

```
    </xsd:complexType>
    <xsd:complexType name="typename">
      <xsd:sequence>
        <xsd:element minOccurs="0"
                     name="extra_information"
                     type="extra" />
        <xsd:element minOccurs="0" name="classname">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element minOccurs="0"
                           name="extra_information"
                           type="extra" />
              <xsd:element name="identifier"
                           type="identifier" />
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="identifier"
                     type="identifier" />
      </xsd:sequence>
    </xsd:complexType>
    <xsd:group name="functiontype">
      <xsd:choice>
        <xsd:element name="partialfunctiontype"
                     type="partialfunctiontype" />
        <xsd:element name="totalfunctiontype"
                     type="totalfunctiontype" />
      </xsd:choice>
    </xsd:group>
    <xsd:complexType name="partialfunctiontype">
      <xsd:sequence>
        <xsd:element minOccurs="0"
                     name="extra_information"
                     type="extra" />
        <xsd:element name="fndom">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:group minOccurs="0" ref="type" />
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="fnrng">
          <xsd:complexType>
```

```
            <xsd:sequence>
              <xsd:group ref="type"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="totalfunctiontype">
      <xsd:sequence>
        <xsd:element minOccurs="0"
                     name="extra_information"
                     type="extra"/>
        <xsd:element name="fndomain">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:group minOccurs="0" ref="type"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="fnrng">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:group ref="type"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
    <!--
=============================================================
VALUE DEFINITIONS
=============================================================
-->
    <xsd:element name="valuedefinition">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element minOccurs="0"
                       name="extra_information"
                       type="extra"/>
          <xsd:group ref="pattern"/>
          <xsd:group minOccurs="0" ref="type"/>
          <xsd:group ref="expression"/>
        </xsd:sequence>
```

```
      <xsd:attribute ref="static" use="optional"/>
      <xsd:attribute ref="access" use="optional"/>
    </xsd:complexType>
  </xsd:element>
  <!--
=============================================================
FUNCTION DEFINITIONS
=============================================================
-->
  <xsd:group name="functiondefinition">
    <xsd:choice>
      <xsd:element ref="explfndef"/>
      <xsd:element ref="implfndef"/>
      <xsd:element ref="extexplfndef"/>
    </xsd:choice>
  </xsd:group>
  <xsd:element name="explfndef">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element minOccurs="0"
                     name="extra_information"
                     type="extra"/>
        <xsd:element name="identifier"
                     type="identifier"/>
        <xsd:element maxOccurs="unbounded"
                     minOccurs="0"
                     name="typevariable"/>
        <xsd:group ref="functiontype"/>
        <xsd:group maxOccurs="unbounded"
                   minOccurs="0"
                   ref="pattern"/>
        <xsd:group ref="functionbody"/>
        <xsd:element minOccurs="0"
                     name="precondition">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element minOccurs="0"
                           name="extra_information"
                           type="extra"/>
              <xsd:group ref="expression"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
```

```
        <xsd:element minOccurs="0"
                     name="postcondition">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element minOccurs="0"
                           name="extra_information"
                           type="extra"/>
              <xsd:group ref="expression"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
      <xsd:attribute ref="static" use="optional"/>
      <xsd:attribute ref="access" use="optional"/>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="implfndef">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element minOccurs="0"
                     name="extra_information"
                     type="extra"/>
        <xsd:element name="identifier"
                     type="identifier"/>
        <xsd:element maxOccurs="unbounded"
                     minOccurs="0"
                     name="typevariable"/>
        <xsd:element maxOccurs="unbounded"
                     name="parametertype"
                     type="pattypepair"/>
        <xsd:element maxOccurs="unbounded"
                     name="identifiertypepair"
                     type="identifiertypepair"/>
        <xsd:element minOccurs="0"
                     name="precondition">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element minOccurs="0"
                           name="extra_information"
                           type="extra"/>
              <xsd:group ref="expression"/>
            </xsd:sequence>
          </xsd:complexType>
```

```
        </xsd:element>
        <xsd:element name="postcondition">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element minOccurs="0"
                           name="extra_information"
                           type="extra"/>
              <xsd:group ref="expression"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
      <xsd:attribute ref="static" use="optional"/>
      <xsd:attribute ref="access" use="optional"/>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="extexplfndef">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element minOccurs="0"
                     name="extra_information"
                     type="extra"/>
        <xsd:element name="identifier"
                     type="identifier"/>
        <xsd:element maxOccurs="unbounded"
                     minOccurs="0"
                     name="typevariable"
                     type="typevariable"/>
        <xsd:element maxOccurs="unbounded"
                     name="parametertypes"
                     type="pattypepair"/>
        <xsd:element maxOccurs="unbounded"
                     name="identifiertypepair"
                     type="identifiertypepair"/>
        <xsd:group ref="functionbody"/>
        <xsd:element minOccurs="0"
                     name="precondition">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element minOccurs="0"
                           name="extra_information"
                           type="extra"/>
              <xsd:group ref="expression"/>
```

```
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
        <xsd:element minOccurs="0"
                     name="postcondition">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element minOccurs="0"
                           name="extra_information"
                           type="extra"/>
              <xsd:group ref="expression"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
      <xsd:attribute ref="static" use="optional"/>
      <xsd:attribute ref="access" use="optional"/>
    </xsd:complexType>
  </xsd:element>
  <xsd:complexType name="pattypepair">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group maxOccurs="unbounded"
                 ref="pattern"/>
      <xsd:group ref="type"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="identifiertypepair">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:element name="identifier"
                   type="identifier"/>
      <xsd:group ref="type"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:group name="functionbody">
    <xsd:choice>
      <xsd:group ref="expression"/>
      <xsd:element name="is_not_yet_specified">
```

```
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element minOccurs="0"
                         name="extra_information"
                         type="extra"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="is_subclass_responsibility">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element minOccurs="0"
                         name="extra_information"
                         type="extra"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:choice>
  </xsd:group>
  <!--
=============================================================
OPERATION DEFINITIONS
=============================================================
-->
  <xsd:group name="operationdefinition">
    <xsd:choice>
      <xsd:element ref="explopdef"/>
      <xsd:element ref="implopdef"/>
      <xsd:element ref="extexplopdef"/>
    </xsd:choice>
  </xsd:group>
  <xsd:element name="explopdef">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element minOccurs="0"
                     name="extra_information"
                     type="extra"/>
        <xsd:element name="identifier"
                     type="identifier"/>
        <xsd:element name="operationtype"
                     type="operationtype"/>
        <xsd:group maxOccurs="unbounded"
                   minOccurs="0"
```

```
                ref="pattern"/>
        <xsd:group ref="operationbody"/>
        <xsd:element minOccurs="0"
                     name="precondition">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element minOccurs="0"
                           name="extra_information"
                           type="extra"/>
              <xsd:group ref="expression"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
        <xsd:element minOccurs="0"
                     name="postcondition">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element minOccurs="0"
                           name="extra_information"
                           type="extra"/>
              <xsd:group ref="expression"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
      <xsd:attribute ref="static" use="optional"/>
      <xsd:attribute ref="access" use="optional"/>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="implopdef">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element minOccurs="0"
                     name="extra_information"
                     type="extra"/>
        <xsd:element name="identifier"
                     type="identifier"/>
        <xsd:element maxOccurs="unbounded"
                     name="parametertype"
                     type="pattypepair"/>
        <xsd:element maxOccurs="unbounded"
                     minOccurs="0"
                     name="identifiertypepair"
```

```
                    type="identifiertypepair"/>
      <xsd:element name="implicitoperationbody"
                    type="implicitoperationbody"/>
    </xsd:sequence>
    <xsd:attribute ref="static" use="optional"/>
    <xsd:attribute ref="access" use="optional"/>
  </xsd:complexType>
</xsd:element>
<xsd:complexType name="implicitoperationbody">
  <xsd:sequence>
    <xsd:element minOccurs="0"
                 name="extra_information"
                 type="extra"/>
    <xsd:element minOccurs="0"
                 name="externals"
                 type="externals"/>
    <xsd:element minOccurs="0" name="precondition">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element minOccurs="0"
                       name="extra_information"
                       type="extra"/>
          <xsd:group ref="expression"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="postcondition">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element minOccurs="0"
                       name="extra_information"
                       type="extra"/>
          <xsd:group ref="expression"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element maxOccurs="unbounded"
                 minOccurs="0"
                 name="error"
                 type="error"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:element name="extexplopdef">
```

```
<xsd:complexType>
  <xsd:sequence>
    <xsd:element minOccurs="0"
                 name="extra_information"
                 type="extra"/>
    <xsd:element name="identifier"
                 type="identifier"/>
    <xsd:element maxOccurs="unbounded"
                 name="parametertype"
                 type="pattypepair"/>
    <xsd:element maxOccurs="unbounded"
                 minOccurs="0"
                 name="identifiertypepair"
                 type="identifiertypepair"/>
    <xsd:group ref="operationbody"/>
    <xsd:element minOccurs="0"
                 name="externals"
                 type="externals"/>
    <xsd:element minOccurs="0"
                 name="precondition">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element minOccurs="0"
                       name="extra_information"
                       type="extra"/>
          <xsd:group ref="expression"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element minOccurs="0"
                 name="postcondition">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element minOccurs="0"
                       name="extra_information"
                       type="extra"/>
          <xsd:group ref="expression"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element maxOccurs="unbounded"
                 minOccurs="0"
                 name="error"
```

```
                    type="error"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:complexType name="operationtype">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:element name="opdom">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element minOccurs="0"
                         name="extra_information"
                         type="extra"/>
            <xsd:group minOccurs="0" ref="type"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="oprng">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element minOccurs="0"
                         name="extra_information"
                         type="extra"/>
            <xsd:group minOccurs="0" ref="type"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:group name="operationbody">
    <xsd:choice>
      <xsd:group ref="statement"/>
      <xsd:element name="is_not_yet_specified">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element minOccurs="0"
                         name="extra_information"
                         type="extra"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
```

```
      <xsd:element name="is_subclass_responsibility">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element minOccurs="0"
                         name="extra_information"
                         type="extra" />
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:choice>
  </xsd:group>
  <xsd:complexType name="externals">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra" />
      <xsd:element maxOccurs="unbounded"
                   name="varinformation"
                   type="varinformation" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="varinformation">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra" />
      <xsd:element name="mode">
        <xsd:complexType>
          <xsd:simpleContent>
            <xsd:extension base="mode" />
          </xsd:simpleContent>
        </xsd:complexType>
      </xsd:element>
      <xsd:element maxOccurs="unbounded"
                   name="name"
                   type="name" />
      <xsd:group minOccurs="0"  ref="type" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:simpleType name="mode">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="rd" />
      <xsd:enumeration value="rw" />
```

```
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:complexType name="error">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:element name="identifier"
                   type="identifier"/>
      <xsd:group ref="expression"/>
      <xsd:group ref="expression"/>
    </xsd:sequence>
  </xsd:complexType>
  <!--
=============================================================
INSTANCE VARIABLE DEFINITIONS
=============================================================
-->
  <xsd:group name="instancevariabledefinition">
    <xsd:choice>
      <xsd:element ref="instassigndef"/>
      <xsd:element ref="instanceinv"/>
    </xsd:choice>
  </xsd:group>
  <xsd:element name="instassigndef">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element minOccurs="0"
                     name="extra_information"
                     type="extra"/>
        <xsd:element name="assigndef"
                     type="assigndef"/>
      </xsd:sequence>
      <xsd:attribute ref="static" use="optional"/>
      <xsd:attribute ref="access" use="optional"/>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="instanceinv">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:group ref="expression"/>
      </xsd:sequence>
    </xsd:complexType>
```

```
  </xsd:element>
  <xsd:element name="initstmt">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:group ref="statement" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <!--
=============================================================
SYNCHRONIZATION DEFINITIONS
=============================================================
-->
  <xsd:group name="syncdefinition">
    <xsd:choice>
      <xsd:element ref="permission" />
      <xsd:element ref="mutex" />
    </xsd:choice>
  </xsd:group>
  <xsd:element name="permission">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element minOccurs="0"
                     name="extra_information"
                     type="extra" />
        <xsd:element name="name" type="name" />
        <xsd:group ref="expression" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="mutex">
    <xsd:complexType>
      <xsd:choice>
        <xsd:element name="all">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element minOccurs="0"
                           name="extra_information"
                           type="extra" />
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
        <xsd:element maxOccurs="unbounded"
```

```
                        name="name"
                        type="name" />
      </xsd:choice>
    </xsd:complexType>
  </xsd:element>
  <!--
=============================================================
THREAD DEFINITIONS
=============================================================
-->
  <xsd:group name="threaddefinition">
    <xsd:choice>
      <xsd:element ref="periodic" />
      <xsd:element ref="procedural" />
    </xsd:choice>
  </xsd:group>
  <xsd:element name="periodic">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element minOccurs="0"
                     name="extra_information"
                     type="extra" />
        <xsd:element name="numeral" type="numlit" />
        <xsd:element name="name" type="name" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="procedural">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element minOccurs="0"
                     name="extra_information"
                     type="extra" />
        <xsd:group ref="statement" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <!--
=============================================================
EXPRESSIONS
=============================================================
-->
  <xsd:group name="expression">
```

```
<xsd:choice>
  <xsd:element name="letexpr" type="letexpr"/>
  <xsd:element name="letbestexpr"
               type="letbestexpr"/>
  <xsd:element name="defexpr" type="defexpr"/>
  <xsd:element name="ifexpr" type="ifexpr"/>
  <xsd:element name="casesexpr"
               type="casesexpr"/>
  <xsd:group ref="unaryexpr"/>
  <xsd:element name="binaryexpr"
               type="binaryexpr"/>
  <xsd:element name="allexpr" type="allexpr"/>
  <xsd:element name="existsexpr"
               type="existsexpr"/>
  <xsd:element name="existsuniqueexpr"
               type="existsuniqueexpr"/>
  <xsd:element name="iotaexpr" type="iotaexpr"/>
  <xsd:element name="setenumerationexpr"
               type="setenumerationexpr"/>
  <xsd:element name="seqenumerationexpr"
               type="seqenumerationexpr"/>
  <xsd:element name="setcomprehensionexpr"
               type="setcomprehensionexpr"/>
  <xsd:element name="setrangeexpr"
               type="setrangeexpr"/>
  <xsd:element name="seqcomprehensionexpr"
               type="seqcomprehensionexpr"/>
  <xsd:element name="subseqexpr"
               type="subseqexpr"/>
  <xsd:element name="mapenumerationexpr"
               type="mapenumerationexpr"/>
  <xsd:element name="mapcomprehensionexpr"
               type="mapcomprehensionexpr"/>
  <xsd:element name="tupleconstructorexpr"
               type="tupleconstructorexpr"/>
  <xsd:element name="recordconstructorexpr"
               type="recordconstructorexpr"/>
  <xsd:element name="recordmodifierexpr"
               type="recordmodifierexpr"/>
  <xsd:element name="bracketedexpr"
               type="bracketedexpr"/>
  <xsd:element name="applyexpr"
               type="applyexpr"/>
```

```
      <xsd:element name="fieldselectexpr"
                   type="fieldselectexpr"/>
      <xsd:element name="tupleselectexpr"
                   type="tupleselectexpr"/>
      <xsd:element name="fcttypeinstexpr"
                   type="fcttypeinstexpr"/>
      <xsd:element name="lambdaexpr"
                   type="lambdaexpr"/>
      <xsd:element name="newexpr" type="newexpr"/>
      <xsd:element name="selfexpr"
                   type="selfexpression"/>
      <xsd:element name="threadidexpr"
                   type="threadidexpression"/>
      <xsd:element name="undefinedexpr"
                   type="undefinedexpression"/>
      <xsd:element name="resultexpr"
                   type="resultexpression"/>
      <xsd:group ref="generalisexpr"/>
      <xsd:element name="preconditionexpr"
                   type="preconditionexpr"/>
      <xsd:element name="isofclassexpr"
                   type="isofclassexpr"/>
      <xsd:element name="isofbaseclassexpr"
                   type="isofbaseclassexpr"/>
      <xsd:element name="sameclassexpr"
                   type="sameclassexpr"/>
      <xsd:element name="samebaseclassexpr"
                   type="samebaseclassexpr"/>
      <xsd:element name="actexpr" type="actexpr"/>
      <xsd:element name="activeexpr"
                   type="activeexpr"/>
      <xsd:element name="waitingexpr"
                   type="waitingexpr"/>
      <xsd:element name="reqexpr" type="reqexpr"/>
      <xsd:element name="finexpr" type="finexpr"/>
      <xsd:element name="name" type="name"/>
      <xsd:element name="oldname" type="oldname"/>
      <xsd:element name="symbolicliteral"
                   type="symbolicliteral"/>
    </xsd:choice>
  </xsd:group>
  <xsd:complexType name="selfexpression">
    <xsd:sequence>
```

```
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra" />
      <xsd:element fixed="self"
                   name="value"
                   type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="threadidexpression">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra" />
      <xsd:element fixed="threadid"
                   name="value"
                   type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="undefinedexpression">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra" />
      <xsd:element fixed="undefined"
                   name="value"
                   type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="resultexpression">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra" />
      <xsd:element fixed="RESULT"
                   name="value"
                   type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="isofbaseclassexpr">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra" />
```

```
      <xsd:element name="name"  type="name" />
      <xsd:group  ref="expression" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="isofclassexpr">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra" />
      <xsd:element name="name"  type="name" />
      <xsd:group  ref="expression" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="sameclassexpr">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra" />
      <xsd:group  ref="expression" />
      <xsd:group  ref="expression" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="samebaseclassexpr">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra" />
      <xsd:group  ref="expression" />
      <xsd:group  ref="expression" />
    </xsd:sequence>
  </xsd:complexType>
  <!--
=============================================================
LOCAL BINDINGS EXPRESSIONS
=============================================================
-->
  <xsd:complexType name="letexpr">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra" />
      <xsd:group  maxOccurs="unbounded"
                  ref="localdefinition" />
```

```
      <xsd:group ref="expression"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="letbestexpr">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group ref="bind"/>
      <xsd:element minOccurs="0" name="best">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element minOccurs="0"
                         name="extra_information"
                         type="extra"/>
            <xsd:group ref="expression"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:group ref="expression"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="defexpr">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:element maxOccurs="unbounded" name="def">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element minOccurs="0"
                         name="extra_information"
                         type="extra"/>
            <xsd:group ref="patternbind"/>
            <xsd:group ref="expression"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:group ref="expression"/>
    </xsd:sequence>
  </xsd:complexType>
  <!--
```

*CONDITIONAL EXPRESSIONS*

```
-->
  <xsd:complexType name="ifexpr">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group ref="expression"/>
      <xsd:group ref="expression"/>
      <xsd:element maxOccurs="unbounded"
                   minOccurs="0"
                   name="elseifexpr"
                   type="elseifexpr"/>
      <xsd:group ref="expression"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="elseifexpr">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group ref="expression"/>
      <xsd:group ref="expression"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="casesexpr">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group ref="expression"/>
      <xsd:element maxOccurs="unbounded"
                   minOccurs="0"
                   name="altn"
                   type="casesaltn"/>
      <xsd:group minOccurs="0" ref="expression"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="casesaltn">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
```

```
                    type="extra"/>
      <xsd:group maxOccurs="unbounded"
                 ref="pattern"/>
      <xsd:group ref="expression"/>
    </xsd:sequence>
  </xsd:complexType>
  <!--
=============================================================
UNARY EXPRESSIONS
=============================================================
-->
  <xsd:group name="unaryexpr">
    <xsd:choice>
      <xsd:element name="prefixexpr"
                   type="prefixexpr"/>
      <xsd:element name="mapinverse"
                   type="mapinverse"/>
    </xsd:choice>
  </xsd:group>
  <xsd:complexType name="mapinverse">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group ref="expression"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="prefixexpr">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group ref="expression"/>
    </xsd:sequence>
    <xsd:attribute name="operator"
                   type="unaryoperator"/>
  </xsd:complexType>
  <xsd:simpleType name="unaryoperator">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="+"/>
      <xsd:enumeration value="-"/>
      <xsd:enumeration value="abs"/>
      <xsd:enumeration value="floor"/>
```

```
      <xsd:enumeration value="not" />
      <xsd:enumeration value="card" />
      <xsd:enumeration value="power" />
      <xsd:enumeration value="dunion" />
      <xsd:enumeration value="dinter" />
      <xsd:enumeration value="hd" />
      <xsd:enumeration value="tl" />
      <xsd:enumeration value="len" />
      <xsd:enumeration value="elems" />
      <xsd:enumeration value="inds" />
      <xsd:enumeration value="conc" />
      <xsd:enumeration value="dom" />
      <xsd:enumeration value="rng" />
      <xsd:enumeration value="merge" />
    </xsd:restriction>
  </xsd:simpleType>
  <!--
=============================================================
BINARY EXPRESSIONS
=============================================================
-->
  <xsd:complexType name="binaryexpr">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra" />
      <xsd:group maxOccurs="unbounded"
                 ref="expression" />
    </xsd:sequence>
    <xsd:attribute name="operator"
                   type="binaryoperator"
                   use="required" />
  </xsd:complexType>
  <xsd:simpleType name="binaryoperator">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="+" />
      <xsd:enumeration value="-" />
      <xsd:enumeration value="*" />
      <xsd:enumeration value="/" />
      <xsd:enumeration value="div" />
      <xsd:enumeration value="rem" />
      <xsd:enumeration value="mod" />
      <xsd:enumeration value="&lt;" />
```

```
      <xsd:enumeration value="&lt;=" />
      <xsd:enumeration value="&gt;" />
      <xsd:enumeration value="&gt;=" />
      <xsd:enumeration value="=" />
      <xsd:enumeration value="&lt;&gt;" />
      <xsd:enumeration value="~=" />
      <xsd:enumeration value="or" />
      <xsd:enumeration value="and" />
      <xsd:enumeration value="=&gt;" />
      <xsd:enumeration value="&lt;=&gt;" />
      <xsd:enumeration value="in set" />
      <xsd:enumeration value="not in set" />
      <xsd:enumeration value="subset" />
      <xsd:enumeration value="psubset" />
      <xsd:enumeration value="union" />
      <xsd:enumeration value="\" />
      <xsd:enumeration value="inter" />
      <xsd:enumeration value="^" />
      <xsd:enumeration value="++" />
      <xsd:enumeration value="munion" />
      <xsd:enumeration value="&lt;:" />
      <xsd:enumeration value="&lt;-:" />
      <xsd:enumeration value=":&gt;" />
      <xsd:enumeration value=":-&gt;" />
      <xsd:enumeration value="comp" />
      <xsd:enumeration value="**" />
    </xsd:restriction>
  </xsd:simpleType>
  <!--
=============================================================
QUANTIFIED EXPRESSIONS & IOTA EXPRESSIONS
=============================================================
-->
  <xsd:complexType name="allexpr">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra" />
      <xsd:element name="bindlist" type="bindlist" />
      <xsd:group ref="expression" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="existsexpr">
```

```
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:element name="bindlist" type="bindlist"/>
      <xsd:group ref="expression"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="existsuniqueexpr">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:element name="bindlist" type="bindlist"/>
      <xsd:group ref="expression"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="iotaexpr">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group ref="bind"/>
      <xsd:group ref="expression"/>
    </xsd:sequence>
  </xsd:complexType>
  <!--
=============================================================
SET EXPRESSIONS & SEQUENCE EXPRESSIONS
=============================================================
-->
  <xsd:complexType name="setenumerationexpr">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group maxOccurs="unbounded"
                 minOccurs="0"
                 ref="expression"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="seqenumerationexpr">
    <xsd:sequence>
```

```
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group maxOccurs="unbounded"
                 minOccurs="0"
                 ref="expression"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="setcomprehensionexpr">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group ref="expression"/>
      <xsd:element name="bindlist" type="bindlist"/>
      <xsd:group minOccurs="0" ref="expression"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="seqcomprehensionexpr">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group ref="expression"/>
      <xsd:element name="setbind" type="setbind"/>
      <xsd:group minOccurs="0" ref="expression"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="setrangeexpr">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group ref="expression"/>
      <xsd:group ref="expression"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="subseqexpr">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group ref="expression"/>
```

```
      <xsd:group ref="expression"/>
    </xsd:sequence>
  </xsd:complexType>
  <!--
=============================================================
MAP EXPRESSIONS
=============================================================
-->
  <xsd:complexType name="mapenumerationexpr">
    <xsd:choice>
      <xsd:element maxOccurs="unbounded"
                   name="maplet"
                   type="maplet"/>
      <xsd:element name="emptymaplet">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element minOccurs="0"
                         name="extra_information"
                         type="extra"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:choice>
  </xsd:complexType>
  <xsd:complexType name="maplet">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group ref="expression"/>
      <xsd:group ref="expression"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="mapcomprehensionexpr">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:element name="maplet" type="maplet"/>
      <xsd:element name="bindlist" type="bindlist"/>
      <xsd:group minOccurs="0" ref="expression"/>
    </xsd:sequence>
  </xsd:complexType>
```

```
  <!--
=============================================================
TUPLE CONSTRUCTOR EXPRESSIONS
=============================================================
-->
  <xsd:complexType name="tupleconstructorexpr">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group maxOccurs="unbounded"
                 ref="expression"/>
    </xsd:sequence>
  </xsd:complexType>
  <!--
=============================================================
RECORD EXPRESSIONS
=============================================================
-->
  <xsd:complexType name="recordconstructorexpr">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:element name="name" type="name"/>
      <xsd:group maxOccurs="unbounded"
                 minOccurs="0"
                 ref="expression"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="recordmodifierexpr">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group ref="expression"/>
      <xsd:element maxOccurs="unbounded"
                   name="recordmodification"
                   type="recordmodification"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="recordmodification">
    <xsd:sequence>
```

```
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:element name="identifier"
                   type="identifier"/>
      <xsd:group ref="expression"/>
    </xsd:sequence>
  </xsd:complexType>
  <!--
=============================================================
APPLY EXPRESSIONS
=============================================================
-->
  <xsd:complexType name="applyexpr">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group maxOccurs="unbounded"
                 minOccurs="0"
                 ref="expression"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="bracketedexpr">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group ref="expression"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="fieldselectexpr">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group ref="expression"/>
      <xsd:element name="identifier"
                   type="identifier"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="tupleselectexpr">
    <xsd:sequence>
```

```
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group ref="expression"/>
      <xsd:element name="numeral" type="numlit"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="fcttypeinstexpr">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:element name="name" type="name"/>
      <xsd:group maxOccurs="unbounded" ref="type"/>
    </xsd:sequence>
  </xsd:complexType>
  <!--
=============================================================
LAMBDA EXPRESSIONS
=============================================================
-->
  <xsd:complexType name="lambdaexpr">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:element name="typebindlist"
                   type="typebindlist"/>
      <xsd:group ref="expression"/>
    </xsd:sequence>
  </xsd:complexType>
  <!--
=============================================================
NEW EXPRESSIONS
=============================================================
-->
  <xsd:complexType name="newexpr">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:element name="name" type="name"/>
      <xsd:group maxOccurs="unbounded"
```

```
                    minOccurs="0"
                    ref="expression"/>
    </xsd:sequence>
  </xsd:complexType>
  <!--
=============================================================
IS EXPRESSIONS
=============================================================
-->
  <xsd:group name="generalisexpr">
    <xsd:choice>
      <xsd:element name="isexpr" type="isexpr"/>
      <xsd:element name="typejudgementexpr"
                   type="typejudgementexpr"/>
    </xsd:choice>
  </xsd:group>
  <xsd:complexType name="typejudgementexpr">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group ref="expression"/>
      <xsd:group ref="type"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="isexpr">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:choice>
        <xsd:element name="name" type="name"/>
        <xsd:element name="basictype"
                     type="basictype"/>
      </xsd:choice>
      <xsd:group ref="expression"/>
    </xsd:sequence>
  </xsd:complexType>
  <!--
=============================================================
PRECONDITION EXPRESSION
=============================================================
 -->
```

```
  <xsd:complexType name="preconditionexpr">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group maxOccurs="unbounded"
                 ref="expression"/>
    </xsd:sequence>
  </xsd:complexType>
  <!--
=============================================================
HISTORY EXPRESSIONS
=============================================================
-->
  <xsd:complexType name="actexpr">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:element maxOccurs="unbounded"
                   name="name"
                   type="name"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="finexpr">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:element maxOccurs="unbounded"
                   name="name"
                   type="name"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="activeexpr">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:element maxOccurs="unbounded"
                   name="name"
                   type="name"/>
    </xsd:sequence>
```

```
  </xsd:complexType>
  <xsd:complexType name="reqexpr">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra" />
      <xsd:element maxOccurs="unbounded"
                   name="name"
                   type="name" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="waitingexpr">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra" />
      <xsd:element maxOccurs="unbounded"
                   name="name"
                   type="name" />
    </xsd:sequence>
  </xsd:complexType>
  <!--
=============================================================
NAMES
=============================================================
-->
  <xsd:complexType name="name">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra" />
      <xsd:element minOccurs="0"  name="classname">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element minOccurs="0"
                         name="extra_information"
                         type="extra" />
            <xsd:element name="identifier"
                         type="identifier" />
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="identifier"
```

```
                    type="identifier"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="oldname">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:element name="identifier"
                   type="identifier"/>
    </xsd:sequence>
  </xsd:complexType>
  <!--
=============================================================
STATE DESIGNATORS
=============================================================
-->
  <xsd:group name="statedesignator">
    <xsd:choice>
      <xsd:element name="name"  type="name"/>
      <xsd:element name="fieldreference"
                   type="fieldreference"/>
      <xsd:element name="maporsequencereference"
                   type="maporsequencereference"/>
    </xsd:choice>
  </xsd:group>
  <xsd:complexType name="fieldreference">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group ref="statedesignator"/>
      <xsd:element name="identifier"
                   type="identifier"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="maporsequencereference">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group ref="statedesignator"/>
      <xsd:group ref="expression"/>
```

```
    </xsd:sequence>
  </xsd:complexType>
  <!--
=============================================================
STATEMENTS
=============================================================
-->
  <xsd:group name="statement">
    <xsd:choice>
      <xsd:element name="letstmt" type="letstmt"/>
      <xsd:element name="letbestmt"
                   type="letbestmt"/>
      <xsd:element name="defstmt" type="defstmt"/>
      <xsd:element name="blockstmt"
                   type="blockstmt"/>
      <xsd:group ref="generalassignstmt"/>
      <xsd:element name="ifstmt" type="ifstmt"/>
      <xsd:element name="casesstmt"
                   type="casesstmt"/>
      <xsd:element name="seqforloopstmt"
                   type="seqforloopstmt"/>
      <xsd:element name="setforloopstmt"
                   type="setforloopstmt"/>
      <xsd:element name="indexforloopstmt"
                   type="indexforloopstmt"/>
      <xsd:element name="whileloopstmt"
                   type="whileloopstmt"/>
      <xsd:element name="nondeterministicstmt"
                   type="nondeterministicstmt"/>
      <xsd:element name="callstmt" type="callstmt"/>
      <xsd:element name="specificationstmt"
                   type="specificationstmt"/>
      <xsd:element name="startliststmt"
                   type="startliststmt"/>
      <xsd:element name="startstmt"
                   type="startstmt"/>
      <xsd:element name="returnstmt"
                   type="returnstmt"/>
      <xsd:element name="alwaysstmt"
                   type="alwaysstmt"/>
      <xsd:element name="trapstmt" type="trapstmt"/>
      <xsd:element name="recursivetrapstmt"
                   type="recursivetrapstmt"/>
```

```
      <xsd:element name="exitstmt"  type="exitstmt"/>
      <xsd:element name="errorstmt"
                   type="errorstmt"/>
      <xsd:element name="skipstmt"  type="skipstmt"/>
    </xsd:choice>
  </xsd:group>
  <xsd:complexType name="errorstmt">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:element fixed="error"
                   name="value"
                   type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="skipstmt">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:element fixed="skip"
                   name="value"
                   type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
  <!--
=============================================================
LOCAL BINDING STATEMENTS
=============================================================
-->
  <xsd:complexType name="letstmt">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group maxOccurs="unbounded"
                 ref="localdefinition"/>
      <xsd:group ref="statement"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:group name="localdefinition">
    <xsd:choice>
```

```
      <xsd:element ref="valuedefinition"/>
      <xsd:group ref="functiondefinition"/>
    </xsd:choice>
  </xsd:group>
  <xsd:complexType name="letbestmt">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group ref="bind"/>
      <xsd:element minOccurs="0" name="best">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element minOccurs="0"
                         name="extra_information"
                         type="extra"/>
            <xsd:group ref="expression"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:group ref="statement"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="defstmt">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:element maxOccurs="unbounded"
                   name="equalsdef">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element minOccurs="0"
                         name="extra_information"
                         type="extra"/>
            <xsd:group ref="patternbind"/>
            <xsd:group ref="expression"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:group ref="statement"/>
    </xsd:sequence>
  </xsd:complexType>
```

```
  <!--
=============================================================
LOCAL BINDING STATEMENTS
=============================================================
-->
  <xsd:complexType name="blockstmt">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:element maxOccurs="unbounded"
                   minOccurs="0"
                   name="assigndef"
                   type="assigndef"/>
      <xsd:group maxOccurs="unbounded"
                 ref="statement"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="assigndef">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:element name="identifier"
                   type="identifier"/>
      <xsd:group ref="type"/>
      <xsd:group minOccurs="0" ref="expression"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:group name="generalassignstmt">
    <xsd:choice>
      <xsd:element name="assignstmt"
                   type="assignstmt"/>
      <xsd:element name="multipleassignstmt"
                   type="multipleassignstmt"/>
    </xsd:choice>
  </xsd:group>
  <xsd:complexType name="assignstmt">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group ref="statedesignator"/>
```

```
      <xsd:group ref="expression"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="multipleassignstmt">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:element maxOccurs="unbounded"
                   minOccurs="2"
                   name="assignstmt"
                   type="assignstmt"/>
    </xsd:sequence>
  </xsd:complexType>
  <!--
================================================================
CONDITIONAL STATEMENTS
================================================================
-->
  <xsd:complexType name="ifstmt">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group ref="expression"/>
      <xsd:group ref="statement"/>
      <xsd:element maxOccurs="unbounded"
                   minOccurs="0"
                   name="elseifstmt"
                   type="elseifstmt"/>
      <xsd:element minOccurs="0" name="else">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element minOccurs="0"
                         name="extra_information"
                         type="extra"/>
            <xsd:group ref="statement"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="elseifstmt">
```

```
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group ref="expression"/>
      <xsd:group ref="statement"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="casesstmt">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group ref="expression"/>
      <xsd:element maxOccurs="unbounded"
                   minOccurs="0"
                   name="altn"
                   type="casesstmtaltn"/>
      <xsd:element minOccurs="0" name="others">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element minOccurs="0"
                         name="extra_information"
                         type="extra"/>
            <xsd:group ref="statement"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="casesstmtaltn">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group maxOccurs="unbounded"
                 ref="pattern"/>
      <xsd:group ref="statement"/>
    </xsd:sequence>
  </xsd:complexType>
  <!--
```

---

*LOOP STATEMENTS*

```
==========================================================
-->
  <xsd:complexType name="seqforloopstmt">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group ref="patternbind"/>
      <xsd:group ref="expression"/>
      <xsd:group ref="statement"/>
    </xsd:sequence>
    <xsd:attribute name="reverse" use="required">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="true"/>
          <xsd:enumeration value="false"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:attribute>
  </xsd:complexType>
  <xsd:complexType name="setforloopstmt">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group ref="pattern"/>
      <xsd:group ref="expression"/>
      <xsd:group ref="statement"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="indexforloopstmt">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:element name="identifier"
                   type="identifier"/>
      <xsd:group ref="expression"/>
      <xsd:group ref="expression"/>
      <xsd:element minOccurs="0" name="by">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element minOccurs="0"
```

```
                                name="extra_information"
                                type="extra" />
                    <xsd:group ref="expression" />
                  </xsd:sequence>
                </xsd:complexType>
              </xsd:element>
              <xsd:group ref="statement" />
            </xsd:sequence>
          </xsd:complexType>
          <xsd:complexType name="whileloopstmt">
            <xsd:sequence>
              <xsd:element minOccurs="0"
                           name="extra_information"
                           type="extra" />
              <xsd:group ref="expression" />
              <xsd:group ref="statement" />
            </xsd:sequence>
          </xsd:complexType>
          <!--
=============================================================
NONDETERMINISTIC STATEMENT
=============================================================
-->
          <xsd:complexType name="nondeterministicstmt">
            <xsd:sequence>
              <xsd:element minOccurs="0"
                           name="extra_information"
                           type="extra" />
              <xsd:group maxOccurs="unbounded"
                         ref="statement" />
            </xsd:sequence>
          </xsd:complexType>
          <!--
=============================================================
CALL AND RETURN STATEMENTS
=============================================================
-->
          <xsd:complexType name="callstmt">
            <xsd:sequence>
              <xsd:element minOccurs="0"
                           name="extra_information"
                           type="extra" />
              <xsd:group minOccurs="0"
```

```
                ref="objectdesignator"/>
    <xsd:element name="callstmtname"  type="name"/>
    <xsd:group maxOccurs="unbounded"
                minOccurs="0"
                ref="expression"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:group name="objectdesignator">
  <xsd:choice>
    <xsd:element name="name"  type="name"/>
    <xsd:element name="selfexpr"
                type="selfexpression"/>
    <xsd:element name="newexpr"  type="newexpr"/>
    <xsd:element name="objectfieldreference"
                type="objectfieldreference"/>
    <xsd:element name="objectapply"
                type="objectapply"/>
  </xsd:choice>
</xsd:group>
<xsd:complexType name="objectfieldreference">
  <xsd:sequence>
    <xsd:element minOccurs="0"
                name="extra_information"
                type="extra"/>
    <xsd:group ref="objectdesignator"/>
    <xsd:element name="identifier"
                type="identifier"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="objectapply">
  <xsd:sequence>
    <xsd:element minOccurs="0"
                name="extra_information"
                type="extra"/>
    <xsd:group ref="objectdesignator"/>
    <xsd:group maxOccurs="unbounded"
                minOccurs="0"
                ref="expression"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="returnstmt">
  <xsd:sequence>
    <xsd:element minOccurs="0"
```

```
                    name="extra_information"
                    type="extra"/>
      <xsd:group minOccurs="0" ref="expression"/>
    </xsd:sequence>
  </xsd:complexType>
  <!--
=============================================================
SPECIFICATION STATEMENT
=============================================================
-->
  <xsd:complexType name="specificationstmt">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                    name="extra_information"
                    type="extra"/>
      <xsd:element name="implicitoperationbody"
                    type="implicitoperationbody"/>
    </xsd:sequence>
  </xsd:complexType>
  <!--
=============================================================
START AND START LIST STATEMENT
=============================================================
-->
  <xsd:complexType name="startstmt">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                    name="extra_information"
                    type="extra"/>
      <xsd:group ref="expression"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="startliststmt">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                    name="extra_information"
                    type="extra"/>
      <xsd:group ref="expression"/>
    </xsd:sequence>
  </xsd:complexType>
  <!--
=============================================================
EXCEPTION HANDLING STATEMENT
```

```
=============================================================
-->
  <xsd:complexType name="alwaysstmt">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group ref="statement"/>
      <xsd:group ref="statement"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="trapstmt">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group ref="patternbind"/>
      <xsd:group ref="statement"/>
      <xsd:group ref="statement"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="recursivetrapstmt">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:element maxOccurs="unbounded"
                   name="trap"
                   type="trap"/>
      <xsd:group ref="statement"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="trap">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group ref="patternbind"/>
      <xsd:group ref="statement"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="exitstmt">
    <xsd:sequence>
```

```
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group minOccurs="0" ref="expression"/>
    </xsd:sequence>
  </xsd:complexType>
  <!--
=============================================================
PATTERNS
=============================================================
-->
  <xsd:group name="pattern">
    <xsd:choice>
      <xsd:element name="patternidentifier"
                   type="patternidentifier"/>
      <xsd:element name="matchvalue"
                   type="matchvalue"/>
      <xsd:group ref="setpattern"/>
      <xsd:group ref="seqpattern"/>
      <xsd:element name="tuplepattern"
                   type="tuplepattern"/>
      <xsd:element name="recordpattern"
                   type="recordpattern"/>
    </xsd:choice>
  </xsd:group>
  <xsd:complexType name="patternidentifier">
    <xsd:choice>
      <xsd:element name="identifier"
                   type="identifier"/>
      <xsd:element name="skip">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element minOccurs="0"
                         name="extra_information"
                         type="extra"/>
            <xsd:element fixed="skip"
                         name="value"
                         type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:choice>
  </xsd:complexType>
```

```
<xsd:complexType name="matchvalue">
  <xsd:choice>
    <xsd:group ref="expression"/>
  </xsd:choice>
</xsd:complexType>
<xsd:group name="setpattern">
  <xsd:choice>
    <xsd:element name="setenumpattern"
                 type="setenumpattern"/>
    <xsd:element name="setunionpattern"
                 type="setunionpattern"/>
  </xsd:choice>
</xsd:group>
<xsd:complexType name="setenumpattern">
  <xsd:sequence>
    <xsd:element minOccurs="0"
                 name="extra_information"
                 type="extra"/>
    <xsd:group maxOccurs="unbounded"
               minOccurs="0"
               ref="pattern"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="setunionpattern">
  <xsd:sequence>
    <xsd:element minOccurs="0"
                 name="extra_information"
                 type="extra"/>
    <xsd:group ref="pattern"/>
    <xsd:group ref="pattern"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:group name="seqpattern">
  <xsd:choice>
    <xsd:element name="seqenumpattern"
                 type="seqenumpattern"/>
    <xsd:element name="seqconcpattern"
                 type="seqconcpattern"/>
  </xsd:choice>
</xsd:group>
<xsd:complexType name="seqenumpattern">
  <xsd:sequence>
    <xsd:element minOccurs="0"
```

```
                    name="extra_information"
                    type="extra" />
      <xsd:group maxOccurs="unbounded"
                 minOccurs="0"
                 ref="pattern" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="seqconcpattern">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra" />
      <xsd:group ref="pattern" />
      <xsd:group ref="pattern" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="tuplepattern">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra" />
      <xsd:group maxOccurs="unbounded"
                 minOccurs="2"
                 ref="pattern" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="recordpattern">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra" />
      <xsd:element name="name"  type="name" />
      <xsd:group maxOccurs="unbounded"
                 minOccurs="0"
                 ref="pattern" />
    </xsd:sequence>
  </xsd:complexType>
  <!--
=============================================================
BINDINGS
=============================================================
-->
  <xsd:group name="patternbind">
```

```
    <xsd:choice>
      <xsd:group ref="pattern"/>
      <xsd:group ref="bind"/>
    </xsd:choice>
  </xsd:group>
  <xsd:group name="bind">
    <xsd:choice>
      <xsd:element name="setbind" type="setbind"/>
      <xsd:element name="typebind" type="typebind"/>
    </xsd:choice>
  </xsd:group>
  <xsd:complexType name="setbind">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group ref="pattern"/>
      <xsd:group ref="expression"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="typebind">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group ref="pattern"/>
      <xsd:group ref="type"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="bindlist">
    <xsd:sequence>
      <xsd:element minOccurs="0"
                   name="extra_information"
                   type="extra"/>
      <xsd:group maxOccurs="unbounded"
                 ref="multiplebind"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:group name="multiplebind">
    <xsd:choice>
      <xsd:element name="multiplesetbind"
                   type="multiplesetbind"/>
      <xsd:element name="multipletypebind"
```

```
                    type="multipletypebind" />
      </xsd:choice>
    </xsd:group>
    <xsd:complexType name="multiplesetbind">
      <xsd:sequence>
        <xsd:element minOccurs="0"
                     name="extra_information"
                     type="extra" />
        <xsd:group maxOccurs="unbounded"
                   ref="pattern" />
        <xsd:group ref="expression" />
      </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="multipletypebind">
      <xsd:sequence>
        <xsd:element minOccurs="0"
                     name="extra_information"
                     type="extra" />
        <xsd:group maxOccurs="unbounded"
                   ref="pattern" />
        <xsd:group ref="type" />
      </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="typebindlist">
      <xsd:sequence>
        <xsd:element minOccurs="0"
                     name="extra_information"
                     type="extra" />
        <xsd:element maxOccurs="unbounded"
                     name="typebind"
                     type="typebind" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:schema>
```

# Appendix D

# Obtaining the Overture Development Tools

The software that has been created as part of this project, can be downloaded from the Internet. For this download it is necessary to have a version of Eclipse (preferably the SDK edition of version 3.0 milestone build 9 or later) already installed on your system.
Using Eclipse, it is possible to download the tool set using the Update Manager. In order to download the software, proceed as follows:

1. Go to Help > Software Updates > Find and Install;
2. Select "Search for new features to install" and press "Next";
3. Press "Add Update Site";
4. Enter "Overture" as the name and "`http://www.overturetool.org/updatesite`" as URL and press "OK";
5. Check the box in front of "Overture" and uncheck "Ignore features ...", then press "Next";
6. Check the box in front of "Overture Feature" and press "Next";
7. From here on you can follow the instructions provided by the Update Manager.

# List of Acronyms

| | |
|---|---|
| API | Applications Programming Interface |
| BNF | Backus-Naur Form |
| CFG | Context-Free Grammar |
| DOM | Document Object Model |
| EBNF | Extended Backus-Naur Form |
| GUI | Graphical User Interface (of a computer program) |
| JavaCC | Java Compiler Compiler |
| JTB | Java Tree Builder |
| LL(k) | A class of language grammars, which can be parsed without backtracking. The first L stands for Left-to-right scan, the second for Leftmost derivation. |
| ODT | Overture Development Tools |
| OML | Overture Modeling Language |
| SableCC | Sable Compiler Compiler |
| SAX | Simple API for XML |
| UI | User Interface (of a computer program); See also GUI |
| XML | Extensible Markup Language |
| XMI | XML Metadata Interchange format |
| W3C | World Wide Web Corporation |
| WXS | W3C XML Schema Language |