DEPARTMENT OF COMPUTER SCIENCE
FACULTY OF SCIENCE
AARHUS UNIVERSITY

TOWARDS DEVELOPMENT OF OVERTURE/VDM++ TO JAVA CODE GENERATOR

BY

MAIHEMUTIJIANG MAIMAITI

20110531

UPPSALA
UNIVERSITET

MASTER'S THESIS

SUPERVISOR: PROF. PETER GORM LARSEN

Department of Computer Science

Aarhus, May 2011

Layout and typography by the author using LaTeX

Abstract

User requirements are usually stated in an informal way in the form of a natural language and graphical illustrations. A good modelling method enables software developers to formalize the requirements in a better way so that the ideas behind the envisaged system can be validated before implementing it in a programming language. As a formal method, VDM++ models the system in mathematically strict (formal) way and uses rigorous notation which has the desired features and gives the possibility that part of the implementation process of the system in a programming language can be automated. The automatic code generation can potentially reduce the cost of implementation. Furthermore, the generated code will then have a direct and strong conformance to its high-level specification (a VDM++ model) which has strong correspondence with the user requirements. Since VDM++ is object-oriented, it is reasonable to generate codes in an object-oriented programming language. In this context, Java is chosen to be the implementation programming language for it is currently one of the most prominent object-oriented languages.
This thesis investigates mapping possibilities from VDM++ constructs to Java and describes the design and development of a Java code generator from VDM++ specification.

Acknowledgements

I would like to thank all of my colleges in this group. First of all, I am very grateful to Professor Peter Gorm Larsen for giving me the opportunity to come to Aarhus University and work in this excellent group on this fantastic topic! I am very thankful to Professor Roland Bol in Uppsala University who made it easy for me to come to Denmark to work on this thesis, and really appreciate all his support and help on my study in Sweden. And I am also thankful to Augusto and Kenneth for the guidance and answering my questions patiently. I am very thankful to the research coordinator Sara for helping me out on lots of things.

Acknowledgments to my parents in Uyghur (My mother tongue): *Ata-anamning we Ailemdiki barliq tuqqanlarning mini yiqindin qollap we ilhamlandurup, yolumni yourutup bergenlikige we men uchun toxtimay qilghan du'alirigha koptin kop rexmet! Mining hemme ishim silerning qollishinglarsiz, silerning du'ayinglarsiz ishqa ashmayti, Allah umringlarni uzun tininglarni salamet qilsun!*

My very special thanks go to my girlfriend Ajigul for everything she did for me; for being with me in all tough and good times, without her supporting and helping me all the way down, this thesis would not be possible for me to finish, Thank you!

Contents

Chapter 1

Introduction

This chapter explains the purpose of this thesis with points to the necessary background knowledge and determines objectives of the thesis. In addition, it gives a brief introduction on the thesis structure with essential reading guidelines.

1.1. Background

With the rapid development of IT industry, software is influencing every aspect of people's life. In the meanwhile, as the demands on software are constantly increasing, requirements on software products are getting more and more complex and volatile. User requirements are usually stated in an informal way in the form of a natural language and graphical illustrations. A good modelling method enables software developers to formalize the requirements in a better way so that the ideas behind the envisaged system can be validated before implementing it in a programming language. This may hopefully reduce the costs of reworking by detecting the defects of the design at the early stage of system development and control the complexity of the system in a more appropriate fashion. Among many popular modelling methods, the Vienna Development Method (VDM) is one of the longest-established and most mature formal method (see [7] for more information about formal methods) for the development of computer-based systems from which the VDM Specification Language (VDM-SL), a formal specification language, is originated [27]. VDM-SL has 2 extensions, respectively VDM++ and VDM-RT which supports the modelling of real time systems. VDM++ is an object-oriented modelling language which uses formal, mathematically-strict notation and also has the desired features described above. A more detailed description of VDM++ can be found in the VDM++ book [9]. Tool support for VDM++ includes both commercial and open-source tools for analyzing models. VDMTools are the leading commercial tools for both VDM and VDM++, owned, marketed, maintained and developed by CSK Systems, building on earlier versions developed by the Danish Company IFAD [4]. In addition, there is an international open-source project called Overture aimed at developing a comprehensive tool set for all VDM dialects (VDM-SL, VDM++, VDM-RT) on the Eclipse platform (for more information see [20]). This thesis is a part of the Overture Project aiming at developing an automatic code generator which automatically generates code (in this case in Java) from VDM++ models which is the high-level specification of the system to be developed.

1.2. Motivation

As a formal method, VDM++ models the system in mathematically strict (formal) way and uses rigorous notation which gives the possibility that part of the implementation process of the system in a programming language can be automated. The automatic code generation can potentially reduce the cost of implementation. Furthermore, the generated code will then have a direct and strong conformance to its high-level specification (a VDM++ model) which has strong correspondence with the user requirements. That is to say that if the user requirements changes, the changes in the implementation can be done automatically and easily which reduces the costs of re-engineering further. Since VDM++ is object-oriented, it is reasonable to generate codes in an object-oriented programming language. In this context, Java is chosen to be the implementation programming language for it is currently one of the most prominent object-oriented languages.

1.3. Thesis Objectives

Main objective of this thesis is to investigate mapping possibilities from VDM++ to Java in a type-safe way to determine the extend at which it is possible to automatically generate compilable and type-safe Java code from VDM++ constructs. More specifically, this thesis aims to determine to which degree the VDM++ classes can be translated to Java classes type-safely.
Before the exploration of mapping possibilities, it is essential to conduct a study on syntax and semantics of both of VDM++ and Java and do the necessary analysis and comparison on them to determine the subset of VDM++ constructs that are transformable to Java. This subset will be the domain of the mapping from VDM++ to Java. As the domain of the map (VDM++ to Java) is determined, the potential ranges (of the map) or potential mapping strategies for each of the identified VDM++ constructs will then be determined. By analyzing and examining the mapping strategies of each of the identified VDM++ constructs in terms of their feasibility and type-safety of the generated Java code, the best suitable mapping strategies for each of the VDM++ constructs will be identified. The following subgoal is to state the determined mapping strategies as a set of transformation rules in a natural language (in this case, English) and subsequently to define them formally in Java. The ultimate result of this thesis is to develop a tool, which is able to generate type-safe Java code from VDM++ constructs, incorporating the formulated transformation rules.
Due to the differences in nature of Java and VDM++, it is expected that some VDM++ constructs cannot be automatically transformed to semantically equivalent Java code. Instead it is planned in such situations to generate skeleton Java code for those parts with sufficient additional information as comments and leave those part to the users to complete.
In addition, it is worth to mention that this thesis concentrates on the transformation of the sequential part of VDM++ to Java, meaning that the concurrent constructs of VDM++ is to be ignored since this has already been considered in [18].
To sum up, the identified goals and steps necessary to achieve the goal of this thesis are illustrated in Figure 1.1.

1.4. Related Work

The process of generating Java code from a VDM++ model can be seen as model to code transformation, which is from high-level specification (e.g. VDM++) to low level specification (e.g.

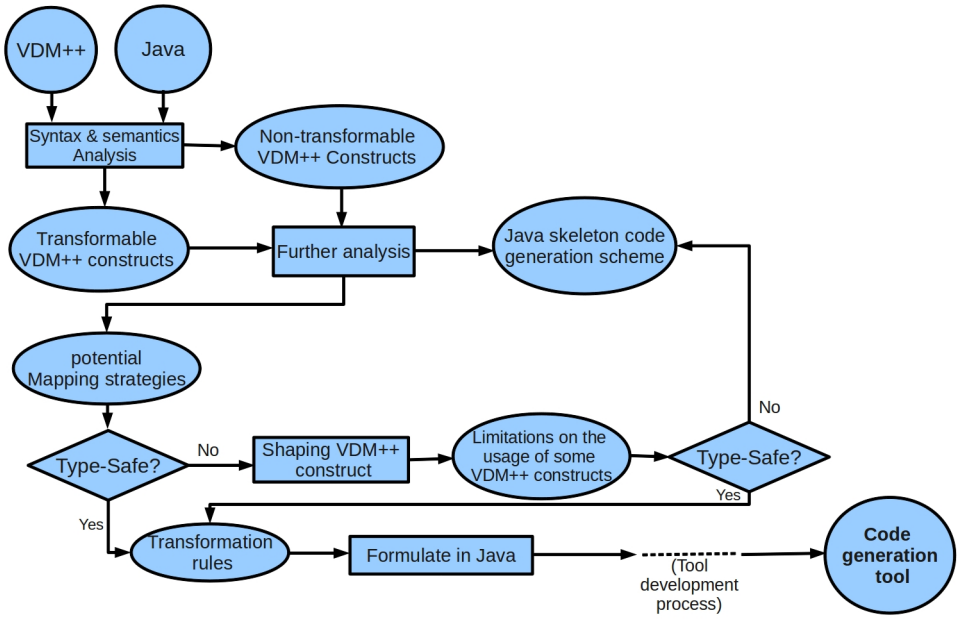

Figure 1.1: Thesis Objectives and Approach Used to Reach the Thesis Objective

Java). The idea of high-level specification to low-level specification transformation is not novel, it has been investigated by others before.

The VDMTools of CSK Systems include a powerful tool - VDM++ to Java code generator, a commercial tool which performs the same task as the result of this thesis intends to. The high robustness of VDMTools VDM++ to Java code generator sacrificed the type-safety of the generated Java code, in addition it is based on relatively old version of Java (J2SE 1.2) which does not have full type-safety feature. In this thesis, it is expected that, with the help of type-safety facilities offered by the newest version of Java (Java SE6), type-safety of the generated code is to be achieved by limiting the use of union type in VDM++ specification.

It should also be noticed that this kind of code generation is normally suitable for formal specifications, since informal specifications are rather ambiguous which makes the definition of transformation rules very hard, even impossible. For instance, design patterns, which are specified informally, are formalized using Balanced Pattern Specification Language (BPSL) (which is a formal method to formally specify patterns) in order to be able to be generated to Java code which is explained by Toufik Taibi and Taieb Mkadmi in [21].

In order to enable such kind of transformation, rules must be defined to enable mapping between 2 languages. Transformation rules can be specified formally in the chosen language (in our case, Java). As for the automatic code generation process, in [2], two different approaches are distinguished in terms of the distance between the formal specification language (in our case, VDM++) and implementation language (in our case, Java). The first one is the long-distance transformation in which the high-level specification is entirely translated into code. Second one is the short-distance one in which the high-level specification is embedded in the implementation code as special comments with only special symbols are translated into implementation code [2]. The code generator PADL2Java, which is a Java Code Generator for Process Algebraic Architectural Descriptions Language, in [2] proposed a long-distance code generation approach for translating PADL descriptions into Java code, since the rather big difference or distance between PADL and

Java.

1.5. Reading Guidelines and Conventions

It is necessary to explain the conventions commonly used in this thesis to ease the reading. Following is the explanation about how Java and VDM++ source code are displayed and how the keywords are highlighted.
Keywords are shown in boldface, for instance, **`int`**.
Representation box for placing VDM++ code:

```
class A
values
public i : int = 3;
operations
public run: nat ==> nat
run(j) == i + 3;
end A
```

Representation box for placing Java code:

```
public class C implements B {
public int foo(int j) {return j+2;}
}
```

Transformation rules are placed in following box:

> **Rule 1:** VDM++ classes are mapped as Java public classes. In addition, noting that the equality between objects in VDM++ only refers to the object references, the inherited `equals` method in the corresponding Java class will be overwritten so that it only checks if the objects are instantiated from the same class.

1.6. Thesis Overview and Structure

This thesis consists of 8 chapters. First part of this thesis, which includes chapter 1, chapter 2 and chapter 3, is mainly introduction and background knowledge which assists reader with the understanding of this work.

Chapter 1 gives a concise introduction about the whole thesis.

Chapter 2 introduces VDM++ briefly and explains important VDM++ constructs.

Chapter 3 explains Java concepts which are essential in understanding the following chapters.

Second part, which includes chapter 4, 5 and 6, is dedicated on the most important part of this thesis which is the illustration on transformation from VDM++ to Java and the limitations.

Chapter 4 gives introduction about the transformation rules for mapping VDM++ classes and types to Java.

Chapter 5 gives detailed introduction about the transformation rules for mapping VDM++ functionality, including instance variable definition, value definition, operation, expression, etc., to Java.

Chapter 6 illustrates the whole transformation process by explaining each component's function and the structure of the code generator.

Chapter 7 explains the limitation of the code generator and challenges faced in transformation process.

The rest of the thesis, chapter 8 is about the conclusion of the thesis.

Appendix A contains information about the abbreviations used in this thesis.

Appendix B contains the complete specification of the VDMJ AST.

Appendix C contains the entire specification of the Java AST.

Appendix D contains the documentation of the classes included in Code Generator Utilities package.

The sourcecode is on CD.

Chapter 2

The VDM Technology

In this chapter, some background knowledge about VDM technology which is essential in understanding of this thesis is presented. This includes a description of the existing tool support for VDM.

2.1. VDM++

The Vienna Development Method (VDM) which is one of the oldest formal modelling methods for system development originating in work conducted at the IBM Development Laboratories in Vienna in the mid 1970s [16]. The core of VDM is VDM Specification Language (VDM-SL) which is a formal modelling language. VDM-SL was standardized by ISO in 1996 [ISOVDM96] and it has two additional dialects, respectively VDM++ and VDM-RT. Thorough introduction about programming in VDM-SL is presented in [8]. VDM-RT supports the modelling of real-time systems, while VDM++ is the object-oriented version of VDM-SL. VDM++ models are consisted of a set of class definitions and objects are the instances of classes. A class is composed of different definition blocks, respectively *value definitions, type definitions, instance variables, operation definitions, function definitions* and *thread definitions*, that can be empty and can be placed in arbitrary order. It is worthwhile to give a brief introduction to each of those definition blocks.

Value-Definitions: Definition of constant values which are similar to the static final variables in Java. A value can be seen as a non-updatable class variable common for all instances of this class.

Type-Definitions: Types refer to either basic types or compound types; basic types refer to boolean types, numeric types, character types, quote types, token types while compound types refer to set types, sequence types, union type, product type, etc., which are constructed using other types by the different type constructors [5]. Type definitions associate type names with type [9]. A type definition block starts with the keyword **types**. For example, a union type can be declared as Listing 2.1.

```
1 class RGB
2 types
3 RGBColor = <Red>|<Green>|<Black>
4 end RGB
```

Listing 2.1: a union type example

In this example, in class `RGB`, a union type `RGBColor`, consisted of `<Red>,<Green>,<Black>` (quote types), is defined, indicating that a value from the `RGBColor` type can be one of those three types.

Instance-Variables: These are the attributes of the objects instantiated from the class. Like Java, every instance variable is associated with a type [9].

Functions: A function takes parameters and returns a result, and in addition functions are not allowed to make use of any object variables or define local variables which implies that they have no side effects. The function body consists of an expression. Listing 2.2 is the example of a very simple function which takes an integer and yields a value that is one larger.

```
1 class A
2 functions
3 public  sum: nat -> nat
4 sum(i) == i+1;
5 end A
```

Listing 2.2: a function example

Operations: Operations are same as functions, except they can use and update instance variables and they are allowed to define local variables which means that they can have side effects. The operations body is composed of statements. Listing 2.3 is an example of a very simple operation which takes an integer `i` and updates the instance variable `n` by adding by input value `i`, so apparently the operation `add` has side effect, because it updated the instance variable `n`.

```
1 class Simple
2 instance variable
3 public static n: int := 1;
4 operations
5 public  add: int ==> nat
6 add(i) ==
7 n = n + i;
8 end Simple
```

Listing 2.3: an operation example

Since this thesis focuses on the code generation of serial part of VDM++, thread definition and synchronization constraints are omitted here.
In VDM++, Class inheritance is denoted by the keyword **`is subclass of`** and multiple inheritance[1] is allowed. Access modifiers, which are respectively **`private`**, **`protected`** and

[1]This is not supported by Java

public, indicate the visibility of the members in the class. Members can also be declared as **static** so that they could be reachable directly through referencing the class.

- **private** members of the class are visible only within the class.
- **protected** members of the class are visible among subclasses of the class.
- **public** members are visible to all classes in the model.

These concepts are discussed further in the mapping (from VDM++ to Java) process. Detailed and thorough introduction about VDM++ concepts and programming is provided in [9] and VDM++ language manual [14].

2.2. Tool Support

Multiple tools exist to support VDM and the different VDM dialects. A commercial tool called VDMTools developed and maintained by CSK is available [3, 4]. VDMTools supports static semantics checking, automatic mappings between VDM and UML, test coverage analysis and automatic C++/Java code generation which, according to [10], covers 95 percent of the VDM language. The VDM++ to Java code generator are further discussed in section 2.3.
In addition, there is an open source tool available for VDM known as VDMJ, which supports VDM-SL, VDM++ and VDM-RT, written in Java. It is now part of the Overture project which provides an IDE based on Eclipse (see section 2.4). The VDMJ tool includes a parser, a type checker, an interpreter, a debugger and a proof obligation generator and it is a command line tool, also accessable from Eclipse [1].

2.3. VDMTools: VDM++ to Java Code Generator

The VDM++ to Java Code Generator supports automatic generation of Java code from VDM++ specifications [6]. Since this thesis aims at developing a similar tool which serves for the same purpose, this tool is discussed in more detail in this section.

2.3.1 Overview of the Code Generation Process

The code generation process in VDMTools is carried out in 3 steps. First, the parser parses a collection of VDM files and then produces a VDM++ abstract syntax tree(AST) from the VDM++ specification files. During the process a type checker extracts type information from the AST and annotates the AST with such type information. The next step is to transform the VDM++ AST to a Java AST which is the main part of the code generator. Finally, there is a backend which takes care of visiting the generated Java AST nodes and creating Java source file. Figure 2.1 provides a more intuitive overview of the code generation process.

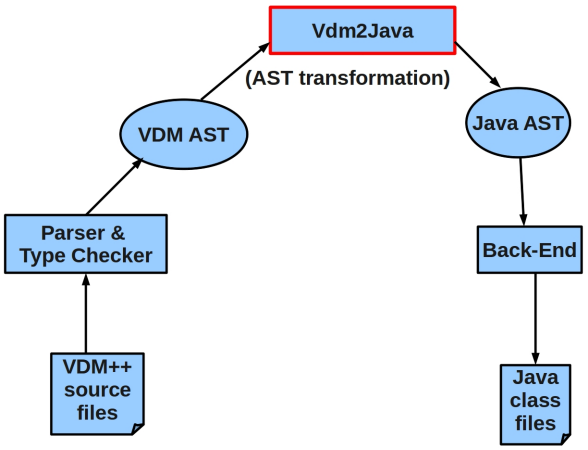

Figure 2.1: Overview of Code Generation

2.3.2 Pros and Cons of the VDMTools Java Code Generator

This section is a brief discussion about the strength and weaknesses of the VDMTools VDM++ to Java code generator. In addition to learning the strength of the VDMTools, this thesis also aims at overcoming some of the weaknesses of VDMTools and make some improvements.

2.3.2.1 Pros

The Code Generator provides a rapid way of implementing Java applications based on VDM++ specifications [6]. Moving on, main advantages of the VDMTools Code Generator will be pointed out.

Robustness It is stated in VDM++ book [9] that 95% of the VDM constructs in VDM++ specifications can be translated to compilable Java code even if there were unsupported constructs in the specification.

Instructiveness Unsupported or omitted constructs and the parts, which need to be completed by user, in the VDM++ specification while generating Java code are informed to the user either by informative comments or error message or warning.

These two are the pros of the Code generator which is directly related to the issues in this thesis, other remarkable strength of the VDMTools code generator can be found in VDM++ book [9] and VDMTools Java Code Generator manual [6].

2.3.2.2 Cons

In addition to the limitations in VDMTools Code Generator which is stated in the Code Generator manual [6], there is a major type-safety in issue in the generated Java code which is one of the main issue and improvement in this thesis.
In VDMTools, type-safety problem is caused mainly by following 2 issues:

Union-type In some cases, VDM++ union types are translated as Object types to Java which need lots of casting when using them in anywhere which may subsequently causes type-safety issues.

Version The version of Java the VDMTools code generator bases on is rather old in which there is no support for Java generics (see [17]). Especially when translating VDM++ collections, the type information of the collection will not be reflected in the generated Java construct, for example, the **`set of int`** type in VDM++ will be translated as **`HashSet`** type in which the type information of the value in `set` is not presented. So this may cause type errors and confusion in the generated Java code.

2.4. The Overture Project

Overture is a community-based open-source project developing the next generation of tools to support modelling and analysis in the design of computer-based systems. Overture project aims at developing a comprehensive tool set for all VDM dialects (VDM-SL, VDM++, VDM-RT) based on the Eclipse platform [20].
This thesis is a part of the Overture Project, the resulting tool of this thesis is to be integrated to Overture IDE. More detailed information about Overture Project can be found in Overture homepage [20].

Chapter 3

The Java Technology

Java is a general-purpose, concurrent, class-based, object-oriented language that is specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere". Java is currently one of the most popular programming languages in use [25].
This chapter gives a general and brief picture about Java and illustrates some concepts which are essential in the context of this thesis. The introduction of Java concepts presented in this chapter is based on the newest version of Java - Java SE 6 (see [19]) .

3.1. Java Classes

As a typical object-oriented language, the building blocks of a Java model are a set of classes. Objects, which store data and provide methods for accessing and modifying this data, plays a main role in a Java program. An object is an instance a class which is made up of definition of attributes (also known as instance variables) and methods [17].

Attributes can be either of instance variables or class variables. Instance variables refer to the non-static variables that store the data of the object while class variables refer to static variables which can be accessed directly referencing the class and can only be modified by the static methods. Attributes can either be of base types, such as integers, booleans, characters, or refer to objects of classes.

Methods act on data. They can be constructors, procedures and functions and they define the behavior of objects from that class [17].

3.1.1 Base Types in Java

Base types in Java which are relevant to this thesis are as following:

`bool` true or false, wrapper class (see [24] for detail) : Boolean

`char` 16-bit Unicode character, wrapper class: Character

`int` 32-bit signed integer, wrapper class: Integer

`long` 64-bit signed integer, wrapper class: Long

`double` 64-bit floating-point number, wrapper class: Double

3.1.2 Modifiers in Java

The modifiers in Java that are involved in this thesis are explained below.

`abstract` The abstract modifier can be used to describe a class or a method. An abstract class is a class that only consists of abstract methods. Abstract methods are declared with the **`abstract`** keyword and are empty.

`static` If a member (can be an attribute or a method) of a Java class is declared with the **`static`** keyword, it means that member of the class can be accessed directly referencing the class.

`final` If an attribute of a Java class is declared with the **`final`** keyword, that attribute can not be modified after initialized. If a class or a method is declared with the **`final`** keyword, that class can not be extended by other classes; that method can not be overwritten in any other subclasses of the class which encompasses the method.

`public` If a field (an attribute) or a method of a class declared with the **`public`** keyword, that field or method is visible to anything in the same package or anything that **`imports`** the class.

`protected` If a field (an attribute) or a method of a class declared with the **`protected`** keyword, that field or method is visible to anything in the same class or subclasses of the class.

`private` If a field (an attribute) or a method of a class declared with the **`private`** keyword, that field or method is only visible within the same class.

Detailed description about the modifiers in Java can be found in [15].

3.1.3 Class Inheritance

In Java, a class can be inherited by other class using **`extends`** keyword, the subclasses will inherit the **`protected`** and **`public`** members of the superclass. And it also should be noticed that one class could only inherit from one superclass (see the Java Language Specification [15] for more detail).

3.2. Important Concepts in Java

This sections introduces some important Java concepts which are crucial in this thesis.

3.2.1 Abstract Class

An abstract class, which is declared with the **`abstract`** keyword, may contain abstract methods as well as concrete methods and instance variables. Abstract classes cannot be instantiated and subclasses of the abstract class should overwrite the abstract methods declared in the abstract class or the subclass will be an abstract class as well. A class can only extend one abstract class.

3.2.2 Java Interfaces

An interface in the Java programming language is an abstract type that is used to specify an interface (in the generic sense of the term) that classes must implement [23]. An interface could be

seen as an **abstract** class which only has **abstract** methods or constants (**static final** attributes), and one class can implement more than one interfaces [15] which inspires a solution for the mapping of VDM++ multiple inheritance structure to Java.

3.2.3 Java Generics

Starting with 5.0, Java introduces a generics framework in which explicit casts are avoided by using abstract types. A generic type is not defined at compilation time but fully specified at run time ([17] or [15]).

3.2.4 Map in Java

HashMap<K,V> indicates an unordered collection of key and value pairs in which key belong to type K, V belongs to type V, each key is associated with one value and all keys are distinct from each other (see [15] for detail).

3.2.5 Set in Java

A Java HashSet<E> is an unordered collection/set of values belonging to same type (E), all values in the set are distinct from each other meaning that there are no same values in the set [15].

3.2.6 Vector

A Vector<E> in Java denotes an ordered and indexed collection of values which belong to type E (can be of any type) and values in it can be the same [15].

3.2.7 Java Math Class

The class Math is Java utility class which includes methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions [19]. The list of the operations and descriptions are listed in the official Java documentation site [19].

3.2.8 Java Operators

In this section, important operators related to this thesis are explained.

instanceof checks if an object is an instance of a *class* or an *interface,* for instance the expression 'a `instanceof` A' returns true if object a is an instance of class A, otherwise returns false. It should be noted that class A could also be the one of superclasses (or interfaces) of the class that constructed object a.

The Java operator above is a bit special and mentioned frequently in the coming chapters, and details about all other operators in Java can be found in Java Language Specification [15].

3.3. Type Safety

Type safety is one of the key issues in this thesis. This section briefly introduces the definition of type safety, why it is important and the type safety mechanism and issues in Java.

3.3.1 General Definition of Type safety

Type safety is the extent to which a programming language discourages or prevents type errors and a type error is erroneous or undesirable program behavior caused by a discrepancy between differing data types. In other words, type-safe code cannot perform an operation on an object that is invalid for that object, for example addition operation ('+' operation) can not be performed on a **`boolean`** type.
Type safety is sometimes alternatively considered to be a property of a computer program rather than the language in which that program is written [26]. Thus, to some extent, type safety largely depends on the programmer himself. However, some programming languages provides type safety facilities and some not. Java language is designed to facilitate type safety.

3.3.2 Type Safety in Java

Java language is designed to enforce type safety which prevents potential type errors, violations and inappropriate-accessing of memory. The type-safety is the most crucial element of Java's security, because every type safety violation creates an opportunity for an untrusted program or application to break out of Java's security restrictions [11]. A very detailed explanation about type-safety in Java can be found in [11] or on this site [12].

3.4. External Libraries Used in The Project

Besides the Java Standard Library, some open source free external Java libraries are used in the project. This section gives a brief introduction about them.

3.4.1 Google Guava Libraries

Guava is a series of open-source core and expanded libraries that include utility classes, google's collections, io classes, and much much more resources aiming at enriching the core JDK 1.6 (or Java SE 6) APIs. For more information, see Guava project homepage [13].
Google's collections provide a very rich operations and some collection types that are not included in Java Standard Library, such as BiMap. Class CGCollections, which is included in a Java utility library called Code Generator Utilities that is built in this project which is compulsory for compiling and executing generated Java code (list of the classes in the library and descriptions about them are presented in Appendix D), is based on google's collections.

Chapter 4

Principles for Mapping VDM++ Classes and Types to Java

In this chapter and chapter 5, the transformation rules and strategies are defined for VDM++ constructs to enable mapping from VDM++ to Java. Each of following sections is dedicated to a group of VDM++ constructs in a top-down fashion. Not all VDM++ constructs can be transformed to Java due to significant differences between the paradigms in the two languages, so these are simply ignored in the mapping process. Omitted constructs are further discussed with more detail and context in each section.
The VDMJ AST (see Appendix B), a top-level VDM++ specification consists of a list of classes. Likewise, a top-level Java specification is also composed by list of classes/interfaces (see Appendix C).
Since the content on principles for mapping all VDM++ constructs to Java is rather substantial, it is presented in 2 Chapters. This chapter is dedicated on the definition of mapping rules for VDM++ classes and types, and mapping rules for the rest of the VDM++ constructs are presented in chapter 5.

4.1. Classes

Since both VDM++ and Java are of object-oriented nature, the notion *class* has same meaning in them. Accordingly, it could be concluded that VDM classes can be translated as Java classes. And since VDM++ classes are accessible within the same model, they are transformed as *public classes* in Java.

> **Rule 1:** VDM++ classes are mapped as Java public classes. In addition, noting that the equality between objects in VDM++ only refers to the object references, the inherited `equals` method in the corresponding Java class will be overwritten so that it only checks if the objects are instantiated from the same class.

4.1.1 Modifiers

Visibility In VDM++, visibility of types, values, instance variables, operations and functions, are expressed by these three modifiers: private, protected, public (see section 2.1), which are also the keywords for defining visibility of class members in Java. Accordingly, here comes transformation rule No.2:

Rule 2: VDM++ visibility modifiers - **`private, protected, public`** are mapped as Java access modifiers - **`private, protected, public`** correspondingly.

static When a member of a VDM++ or a Java class is declared with the **`static`** keyword, it is then a class member which means it can be accessed directly with the class without creating an instance, therefore the '**`static`**' keyword of VDM++ is mapped as **`static`** in Java.

Rule 3: The keyword **`static`** in VDM++ is mapped as **`static`** in Java.

4.1.2 Abstract Classes

In VDM++, if a class contains a delegated definition (see section 2.1), it is called an abstract class and instantiating an object from an abstract class is not allowed [9]. The subclass of an abstract class which does not implement the delegated operations and functions is also abstract.This is similar to the abstract classes (see *section* 3.2.1) in Java, except a Java class containing only abstract methods is termed as an interface (see *section* 3.2.2). Therefore, an abstract VDM++ class could be mapped as a Java abstract class. And a VDM++ abstract class can also be mapped as a Java interface as well, only if it satisfies certain conditions, since Java interface does not contain any instance variables and only contains abstract methods.

Rule 4: All VDM++ abstract classes are mapped as Java abstract classes, except the ones which satisfy the following conditions are mapped as Java interfaces.

1. All containing functions and operations are abstract;
2. There are no instance variables;
3. Values are defined directly (see *section* 5.1)
4. Their superclasses also satisfy the above 3 conditions.

To illustrate the point of this rule, it is necessary to give an example.

```
1  class A
2  values
3  public i : int = 3;
4  operations
5  public run: nat ==> nat
6  run(n)== is subclass responsibility;
7  end A
8
9  class B is subclass of A
10 functions
11 public f: int -> int
```

```
12 f(x) == is subclass responsibility
13 end B
14
15 class C is subclass of B
16 functions
17 public foo: int -> int
18 foo(j) == j+2;
19 end C
```

Listing 4.1: The VDM++ abstract classes

In Listing 4.1, class A contains a delegated operation and a directly defined value, therefore it meets the requirement to be mapped as a Java interface, and its subclass B also could be mapped as Java interface since it contains a delegate function and its superclass is an interface. As for class C, its superclass is an interface, though it contains a function with actual body, as a consequence, class C can be mapped as an abstract class. Accordingly, the VDM++ codes in Listing 4.1 will be Java-code-generated as Listing 4.2.

```
1  public interface A {
2  public static final int i =3;
3  public abstract int run (int n);
4  }
5
6  public interface B extends A {
7  public abstract int f(int x);
8  }
9
10 public abstract class C impliments B {
11 public int foo(int j) {return j+2;}
12 }
```

Listing 4.2: Generated Java classes

4.1.3 Inheritance

The inheritance structure of VDM++ and Java classes are the same, except that VDM++ allows multiple inheritance while, in Java, a class can *implement* multiple interfaces but only can extend one class.
A single VDM++ class inheritance can be mapped as a single Java class inheritance which is denoted by the keyword **extend**. As for multiple inheritance, the mapping is possible when at most one of the superclasses is not transformable to Java interface (see *section* 4.1.2).

> **Rule 5:** Single VDM++ class inheritance is mapped as Java class inheritance which is denoted by the keyword *extend*, as for multiple inheritance, the mapping is possible only when at most one of the superclasses is not transformable to Java interface.

Listing 4.1 and Listing 4.2 are examples of single inheritance. An example (in Listing 4.3), which

continues on the bases of Listing 4.1 & 4.2, is provided for better understanding of the handling of multiple inheritance. Class `B` and `C` refers to the ones in Listing 4.1

```
1  class D
2  operations
3  public op: nat ==> bool
4  op(b)== is subclass responsibility
5  end D
6
7  class E is subclass of B,C,D
8  values
9  public bo: bool = true;
10 end E
```

Listing 4.3: Multiple Inheritance in VDM++

In Listing 4.3, class `E` inherits two abstract classes `B, D` which are transformable to Java interface and one abstract class `C` that can not be mapped as Java interface. Furthermore, among the class `E`'s super classes `B, C, D`, there is only one abstract class `C` which can't be mapped as Java interface, therefore this multiple inheritance could be mapped to Java counterparts as shown in Listing 4.4.

```
1  public interface D {
2  public abstract boolean op(int b);
3  }
4
5  public abstract class E extends C implements B,D {
6  public static final bo: boolean = true;
7  }
```

Listing 4.4: Generated Multi-inheritance in Java

4.2. Types

4.2.1 Base Types

VDM++ numeric types **`real, rat`** and **`int, nat, nat1`** are mapped as Java primitive types **`double`** and **`int`** respectively. Moreover, VDM++ `char` and `bool` types are exactly the same as Java **`char`** and **`boolean`** types.

Rule 6: VDM++ **`real, rat`** types are mapped as Java **`double`** type; VDM++ **`int, nat, nat1`** types are mapped as Java **`int`** type; VDM++ **`char`** and **`bool`** are mapped as Java **`char`** and **`boolean`** types.

The Quote Type consists of a pair of '<' and '>' and a quote literal and the quote type has the same name as the quote literal. It has some resemblance with `enum` type in Java, however there are 2 significant differences between them. Firstly, the quote type is composed of a single quote literal as its only value while Java `enum` has a set of literals or constants; secondly, a quote type's name is the quote literal itself while Java usually have a name which is different from all of its containing constants. As a matter of fact, quote type has more resemblance to a normal Java class, since the class itself represents a type and it is represented by its name (class name), furthermore the `equals` method of a Java class which is inherited from superclass *object* could be overwritten to enable comparison between type instances in the same way as the quote type in VDM++. Thus, the feasible solution is to map a quote type as a Java class.

Rule 7: VDM++ `quote type` is mapped as a *non-abstract* Java `class` with the corresponding quote literal as the class name.

To make the point more clear, the following example Listing 4.5 and 4.6 is presented.

```
1 <RED>
```

Listing 4.5: A quote type in VDM++

The `<RED>` type in Listing 4.5 is code generated as Listing 4.6.

```
1 public class RED {
2 static private int hc = 0;
3 public RED () { if (hc == 0) hc = super.hashCode(); }
4 public int hashCode () { return hc; }
5 public boolean equals (Object obj) { return obj instanceof RED; }
6 public String toString () { return "<RED>"; }
7 }
```

Listing 4.6: Generated Java `class` for the `quote type`

4.2.2 Compound Types

Compound types refer to types that are constructed using other types by the different type constructors [5], such as set types, sequence types, union type, product type, etc. This sections presents the mapping strategies of VDM++ compound types to Java.

4.2.2.1 Set Type

A set in VDM++ is an unsorted collection of values belonging to same type [14]. Luckily, Java offers a comprehensive set facilities (see section 3.2.5), such as TreeSet<E>, HashSet<E>, etc. Since TreeSet (see section 3.2.5) is ordered while HashSet is not, set type of VDM++ is mapped as HashSet in Java.

Rule 8: VDM++ set type is mapped as Java HashSet<E> (E is the type of values in the set).

For example, Listing 4.7 will be translated as Listing 4.8.

```
1 set of int
```

Listing 4.7: A Set in VDM++

```
1 HashSet<Integer>
```

Listing 4.8: Generated Java Code

4.2.2.2 Sequence Types

A sequence in VDM++ is an ordered collection of values belonging to same type [14], and all values are indexed from 1 to the length of the sequence. There is an allmost the same construct in Java that is called Vector (see section 3.2.6). A vector is also an ordered collection of values of same type, so VDM++ sequence type is mapped as Java Vector<E> (E is the type of values in the Vector) type.

Rule 9: VDM++ sequence type is mapped as Java Vector<E>.

For example, Listing 4.9 will be translated as Listing 4.10.

```
1 seq of int
2 seq1 of int
```

Listing 4.9: Sequence Type in VDM++

```
1 Vector<Integer>
2 Vector<Integer>
```

Listing 4.10: Generated Java Code

Thus, the fact that a **seq1** type cannot have empty sequence is ignored in the mapping.

4.2.2.3 Map Types

A map in VDM++ is an unordered collection of pairs; A map type from type A to type B associates each element of A with an element of B (for detail see [14]) and elements of A are distinct from each other. In fact, there is two kinds of maps in VDM++, respectively *general map type* and *Injective map type*. In an Injective map from type A to Type B, elements of B are distinct from each other as well. Fortunately, HashMap (see section 3.2.4) in Java is also an unordered collection

of pairs, so a VDM++ map type (both of general and injective map types) could be simply mapped as HashMap<K,V> (K and V are the types of the pairs in the map) in Java. And thanks to Google collection library (see section 3.4.1), injective map type could be mapped as HashBiMap<K,V> which is defined in Google collection library and of the same nature as VDM++ injective map.

> **Rule 10:** VDM++ general map type is mapped as Java HashMap<K,V>, injective map type is mapped as Java HashBiMap<K,V>.

For example, Listing 4.11 will be translated as Listing 4.12.

```
1 map A to B
2 inmap A to B
```

Listing 4.11: Map in VDM++

```
1 HashMap<A,B>
2 HashBiMap<A,B>
```

Listing 4.12: Generated Java Code

4.2.2.4 Product Types

A product type consists of a fixed combination of other types (for details, see [9]). Product type is not supported in this code generator, so it is ignored here.

4.2.2.5 Record Types

A record type is consisted of other types in such a way that each of those types is a field, which optionally has a distinct identifier, of the record type. Components of a record type are arranged in a fixed order and can be selected directly through its corresponding identifier. Listing 4.13 gives a general form of a record type.

```
1 record ::
2          field 1 : type 1
3          ...
4          field n : type n
```

Listing 4.13: General Form of A Record Type

A record constructor is as Listing 4.14.

```
1 mk_record(value 1, ..., value n)
```

Listing 4.14: Record Constuctor

Detailed description about *record type* can be found in [14]. According to [14], operators can be performed on *record types* are as Table 4.1.

Operator	Name	Type
r . i	Field select	$A * Id \rightarrow Ai$
r1 = r2	Equality	$A * A \rightarrow bool$
r1 <> r2	Inequality	$A * A \rightarrow bool$
is_A(r1)	Is	$Id * Master\ A \rightarrow bool$

Table 4.1: Operators On Record Type

It can be easily seen that characteristics of the *record type* has much resemblance to a Java class. Like a *record type*, a Java class has fields, also known as *instance variables;* each field has an identifier and refers to a *class* or a *base type* (see section 3.1.1); values of fields can be selected through the field identifier. As for the *record constructor,* a constructor, which initializes all instance variables, of a Java class will do the same work. Furthermore, all four type operators (listed in Table 4.1) of a *record type* have equivalent counterparts in Java.

Rule 11: A VDM++ *record type* is mapped as a *Java class* in the following way:

- *Fields* of the record type are mapped as *instance variables* of the Java class, and for those unnamed *fields*, names will be generated automatically for corresponding *instance variables*;
- Record constructor is mapped as the constructor, which initializes all instance variables, of the corresponding Java class.
- Equality/inequality operator is mapped as an overwritten *equals* method within the Java class ensuring that values of the *fields/instance variables* are compared as well (when 2 instances are compared). In addition, fields that are declared using ':-' (see section 2.1) will be ignored in *equals* method.
- `is` operator of the *record type* is mapped as the `instanceof` operator (see section 3.2.8) of Java, for instance, `is_A(r1)` will be mapped as `r1 instanceof A`.

Listing 4.15 will be translated as Listing 4.16.

```
1 class T
2 types
3 public SALE ::  PrId :   seq of char
4                 Prprice :  real;
5 end T
```

Listing 4.15: An example of Record Type

```
1 public class T {
2    public static class SALE {
```

```
3           public String PrId;
4           public double Prprice;
5
6           public SALE () {}
7           public SALE (String p1, double p2) {
8              PrId   = p1;
9              Prprice = p2;
10          }
11          public String toString () {
12            return "mk_T`SALE("+Prname+","+PrId+","+Prprice+")";
13          }
14          public bool equals(Object obj) {
15            if (obj instanceof SALE) {
16              SALE tmp = (SALE) obj;
17              if (this.PrId=tmp.PrId && this.Prprice=tmp.Prprice)
18                  return true;
19              else
20                  return false;
21           }
22           else
23                return false;
24      }
25  }
```

Listing 4.16: Generated Code for Listing 4.15

The record constructor in Listing 4.14 will be translated as Listing 4.17.

```
1  new record(value 1, ..., value n);
```

Listing 4.17: Record Constructor

4.2.2.6 Union Type

A union type is composed by several types and contains all values of each of the components [9]. Normally the component types are *disjoint,* which implies that there are no common elements. It is not recommended to form a union type of non-disjoint types.
A union type is defined as Listing 4.18:

```
1  type 1 | type 2 | type 3 | ... | type n;
```

Listing 4.18: Union Type

Translation of the union type is one of the major challenges in this thesis. A union type could be declared in *type definition* block in standalone manner or specified directly as a component of a more complex type, such as *set type, function type, sequence type, etc.* The union type is the abstraction of several types that the concrete type of its instance will not be known until a value/object assigned to it or until runtime. Unfortunately, there is no similar construct in Java.

Though, it is always possible to define a new type as a class in Java, so it could be a way forward to map the union type as a Java class and map component types of the union type as subclasses of the class. However, it should be noticed that a component type (such as `<red>` in Listing 4.19) of a union type could appear in another union type, which means that the translated class of that component type could inherit more than one Java classes while multiple inheritance is not allowed in Java. As a consequence, it would be wise to translate the union types to Java interfaces (see section 3.2.2 for details) in this special case, because a class can implement more than one interfaces in Java.

```
1 CarColor = <Black> | <Red>;
2 LampColor = <Red> | <White>;
```

Listing 4.19: a union type example

However, there is no general mapping strategy for translating all forms of union types. Firstly, mapping union type that consisted of union types may become impossible in the case as shown in Listing 4.20, `T1` would be transformed as an interface and `T2` will be translated as the subinterface of `T1`, but `T1` is also a component of `T2` which means `T1` will be mapped as the subinterface of `T2`, eventually resulting that `T1` becomes the subclass of itself and this is not allowed in Java.

```
1 T1 = record1 | T2;
2 T2 = record1 | T1;
```

Listing 4.20: Union type of union types

On the other hand, mapping of union types that are consisted of base types excluding *quote type* becomes complicated.There are counter parts of VDM++ base types in Java, so if the union type is mapped as an interface and the elements (which are base types) should be mapped as subclasses of it, while the base types of Java is not allowed to be redefined (to make them become subclasses of the interface which is translated from the union type), and the class versions of base types in Java (see section 3.1.1) such as Integer, Double, Boolean, etc., are final which means that they can not be extended, so a class, which is subclass of a Java base type class (e.g. Integer) and the generated interface, can not be created. Consequently, such union types can not be simply mapped as Java Interfaces. However, there is a simple way of dealing with it. This is the approach exploited in VDMTools VDM++ to Java Code Generator (see section 2.3 for details) in which the union type of base types is mapped as the Object type of Java. However, this approach will result in not type-safe Java code, because Object class is the superclass of all classes in Java meaning that instances of any type or any class are also instances of Object class which is equivalent to a Union type in VDM++ that is consisted of all types that exist in VDM++. For example a function which takes a parameter of union type that consisted of **`bool`** and **`nat`** will only take values either of **`bool`** or **`nat`** types, but after translating the union type to Java as the Object type, the equivalent Java method will accepts any parameter of any types exist in Java, not just **`boolean`** and **`int`**, and this may cause type errors. So apparently this approach sacrifices type-safety (see section 3.3) of the generated Java code. Accordingly, it would be reasonable to limit the use of union types by setting up stricter rules for union types. Before doing that, it is necessary to take a look at a special form of union type. The union types that are consisted of *quote types* and *record types* could be simply mapped as Java interfaces since both of quote types and record types are mapped as Java classes and it is possible that different union types include common quote or record types which could cause multiple inheritance issues.

Rule 12: Union types that are consisted of quote and record types are mapped as Java interfaces and the member quote or record types are mapped as subclasses of corresponding Java interfaces.

For example, a union type named as U containing the quote type <RED> (in Listing 4.5 on page 19) and the record type SALE (in Listing 4.15) would be translated as Listing 4.21.

```
1  public static interface U {}
2  public static class RED implements U {
3         public RED () {}
4         public boolean equals (Object obj) {
5             return obj instanceof RED; }
6         public String toString () { return "<RED>"; }
7  }
8  public static class SALE implements U {
9           public String PrId;
10          public Double Prprice;
11
12          public SALE () {}
13          public SALE (String p1, double p2) {
14             PrId   = p1;
15             Prprice = p2;
16          }
17          public String toString () {
18            return "mk_T`SALE("+Prname+","+PrId+","+Prprice+")";
19          }
20          public boolean equals(Object obj) {
21            if (obj instanceof SALE) {
22              SALE tmp = (SALE) obj;
23              if (this.PrId.equals(tmp.PrId) &&
24                  this.Prprice.equals(tmp.Prprice))
25                  return true;
26              else
27                  return false;
28           }
29           else
30                return false;
31       }
32       }
```

Listing 4.21: Generated Java Code

Since *record type* can contain all kinds of types, union types that are consisted of other types than *quote* and *record types* could be defined as equivalent union types of *record types* instead, in order to be able to be translated to Java. That is to say that types such as *numeric types*, *bool type*, *char type* and *union types, etc.* should be wrapped in the form of *record types* first before consisting a *union type.* For example, the union type Union2 and Union in Listing 4.22 should be defined as Listing 4.23 instead in order to be translated to Java.

```
Union = real | bool;
Union2 = char | Union;
```

Listing 4.22: Union of base types

```
class T
types
Union = R | B;
Union2 = C | U;
R:: r : real;
B:: b : bool;
U:: u : Union;
C:: c : char;
end T
```

Listing 4.23: Union2 and Union after modification

For the sake of simplification and generalization of the mapping strategy of union types, it is necessary to require all types other than *quote type* and *record type* to be wrapped in the form of *record types* before forming a union type. In the meanwhile, direct use of union types in function and operations, e.g. a function which takes parameters of `T1|T2` type, is not allowed as well to avoid generating redundant temporary interfaces for union types and other complexities it may cause.

Rule 13: The elements in a union type, except from *quote* and *record types*, are required to be wrapped in the form of *record types* before forming a union type in order to be mapped to Java. In addition, direct use of union types is not allowed, all union types should be defined in type definition block before being used.

For example, a union type as Listing 4.24 will be translated as Listing 4.25.

```
class T
types
Union = R | B;
Union2 = R | U;
R:: r : real;
B:: b : bool;
U:: u : Union;
end T
```

Listing 4.24: another example of Union Type

```
public class T {
public static interface Union {}
public static interface Union2 {}
```

```
4  public static class R implements Union, Union2 {
5  public Double r;
6
7            public R () {}
8            public R (double d) {
9               r = d;
10           }
11           public String toString () {
12             return "mk_T`R("+r+")";
13           }
14           public bool equals(Object obj) {
15             if (obj instanceof R) {
16               R tmp = (R) obj;
17               if (this.r.equals(tmp.r))
18                   return true;
19               else
20                   return false;
21            }
22            else
23                 return false;
24       }
25 }
26 public static class B implements Union {
27           public Boolean b;
28           public B () {}
29           public B (boolean l) {
30              b = l;
31           }
32           public String toString () {
33             return "mk_T`B("+b+")";
34           }
35           public bool equals(Object obj) {
36             if (obj instanceof B) {
37               B tmp = (B) obj;
38               if (this.b.equals(tmp.b))
39                   return true;
40               else
41                   return false;
42            }
43            else
44                 return false;
45       }
46    }
47 public static class U implements Union2 {
48           public Union u;
49           public U () {}
50           public U (Union l) {
51              u = l;
52           }
```

```
53          public String toString () {
54            return "mk_T`U("+u+")";
55          }
56          public bool equals(Object obj) {
57            if (obj instanceof U) {
58              U tmp = (U) obj;
59              if (this.u.equals(tmp.u))
60                  return true;
61              else
62                  return false;
63             }
64             else
65                  return false;
66       }
67     }
68 }
```

Listing 4.25: Translation of Listing 4.24

4.2.2.7 Optional Type

An optional type `[T]` is equivalent to a union type `T | nil` where **nil** stands for the absence of a value. However, optional type is not supported in the code generator, so it is not further discussed here.

4.2.2.8 The Object Reference Type

The values of an object reference type refer to objects of a given class, and an object reference type is defined using a class name such as type A in Listing 4.26.
It should be noted that the *class name* in the object reference type must be the name of one of the classes defined in the model. The object reference type is mostly used in the definitions of instance variables (see 2.1) in which a variable can be an object reference to the object of a class, and the object can be created by using the *new expression* (see section 5.4). Likewise, in Java, a class name represents a type (see section 3.1) which has the same nature as the object reference type in VDM++. Therefore, a VDM++ object reference type, which is denoted by a class name, is translated as the corresponding class name (which is the same as the original VDM++ class name) in Java.

Rule 14: A VDM++ object reference type, which is denoted by a class name, is mapped as the same class name (as the corresponding VDM++ class name) in Java.

For instance, the variable a which is of object reference type A in Listing 4.26 is generated as the variable a of type A in Listing 4.27 to Java.

```
1 class Z
2 instance variables
3   a: A := new A();
```

```
4    ...
5  end Z
```

Listing 4.26: An Example of An Object Reference Type

```
1  public class Z {
2    A a = new A();
3    ...
4  }
```

Listing 4.27: Generated Java Code for Listing 4.26

4.2.2.9 Function Types

Function type is not supported in the code generator, so it is ignored here.

4.2.3 Type Definition

A type definition block starts with the keyword **types**, followed by an arbitrary number of type definitions (see section 2.1) separated by semicolons as in Listing 4.28.

```
1  class A
2  types
3  type definition 1;
4  type definition 2;
5  ...
6  end A
```

Listing 4.28: Type Definition

in which a type definition has 2 forms as Listing 4.29 and Listing 4.30,

```
1  type_name = type
```

Listing 4.29: A form of Type Definition

or

```
1  record type
```

Listing 4.30: Another form of Type Definition

Mapping of type definitions of the form in Listing 4.30 is actually the same as *record type* definitions which was explained in section 4.2.2.5.
As for the mapping of type definitions of the form in Listing 4.29, it depends on the *type*. If it is union type, it will be dealt in the same way as the *union type definitions* in section 4.2.2.6. And if it is a type other than union type, the type definition will be simply ignored when mapping to Java,

and the type name will be replaced by corresponding type, with which the type name is associated, in all places that the type name appears, since there is no similar construct in Java and it is not necessary to define one.

> **Rule 15:** A type definition associating a type name with a type that other than *union type* is ignored when mapping and the type name will be replaced by the associating type in all places that the type name appears.

For example, Listing 4.31 is translated as Listing 4.32 to Java:

```
1 class A
2 types
3 T1 = nat;
4 T2 = real;
5 functions
6 public run: T1 -> T2
7         run(t) ==
8         t + 0.1;
9 end A
```

Listing 4.31: An Example of Type Definitions

```
1 public class A {
2 public double run(int t) {return t + 0.1;}
3 }
```

Listing 4.32: Generated Java Code

In Listing 4.31, `T1` and `T2` are respectively associated with **`nat`** type and **`real`** type, and they are used in as parameter type and return type in function `run`. However, definitions of `T` and `T2` are ignored in code generation and they are replaced by int and double in the definition of method `run` in Listing 4.32.

Chapter 5

Principles for Mapping VDM++ Functionality to Java

This chapter is the continuation of chapter 4. The rules for mapping VDM++ functionality to Java are presented in detail.

5.1. Values

Value definitions correspond to the constant values definitions in traditional programming languages, such as C, C++. And a constant is equivalent to an initialized static final (see section 3.1.2) variable in Java. Consequently, a VDM++ value definition is mapped as an initialized static final variable definition in Java.

> **Rule 16:** Value definitions are mapped as initialized static final variable definitions in Java.

For instance, value definitions in Listing 5.1 are translated as Listing 5.2 to Java:

```
1 class T
2 values
3 public size: int = 50;
4 protected price: real = 30.23;
5 end T
```

Listing 5.1: Value Definition

```
1 public class T {
2   public static final int size = 50;
3   protected static final double price = 30.23;
4 }
```

Listing 5.2: Generated Java Code for Listing 5.1

5.2. Instance Variables

The internal state of an object or a class is called *instance variables* of the object or the class. If the variable has static access (see section 4.1.1), it refers to the internal state of the class, otherwise the object, and also a VDM++ instance variable definition consists of a name, a type and an optional initial value. It could be noticed that the instance variables in VDM++ are have the same nature as the attributes in Java (see section 3.1). Accordingly an instance variable definition is translated as a definition of attribute to Java.

> **Rule 17:** Instance variable definitions in VDM++ are mapped as definitions of attributes in Java.

For instance, instance variable definitions in Listing 5.3 could be translated as Listing 5.4 to Java.

```
1 class I
2 instance variables
3 public static distance: real := 0.001;
4 private station: seq of char;
5 end I
```

Listing 5.3: Instance Variable Definition

```
1 public class I {
2   public static double distance = 0.001;
3   private String station;
4 }
```

Listing 5.4: Generated Java Code

5.3. Functions & Operations

This section conducts a comparative analysis on VDM++ and Java functionality and subsequently finds out and presents suitable strategies for mapping VDM++ functions and operations to Java methods.
In VDM++, algorithms can be specified by both functions and operations [5]. A function takes input parameters and produces a result with no reference to the instance variables of the object implying that it must always return same result for the same given input [9]. An operation also takes input parameters and produces a result, however it may read and modify instance variables which means that it may produce different result for same input on different objects.

5.3.1 Function Definitions

A function definition can have a *precondition* and a *postcondition,* as well as a function body. A *precondition* is a logical expression stating the conditions under which the modeler expects the

function to be applied while a *postcondition* is a boolean expression which describes the relation between the inputs and result of the function [9]. *Preconditions* and *postconditions* starts respectively with **pre** and **post** keywords within the function definition as shown in Listing 5.5.

```
1 sqrt (x:rat) r: real
2 pre x>= 0
3 post r * r = x
```

Listing 5.5: Pre and post conditions

A function body consists of an expression or can be specified by the keyword '**is not yet specified**' or '**is subclass responsibility**'.
The keyword "**is not yet specified**" indicates that the implementation of this body must be performed by the developer or user, while the keyword "**is not yet specified**" indicates that the implementation of this body must be undertaken by any subclasses [5].
Function definitions can be classified as *implicit* and *explicit* function definitions in terms of the presence of the function body. *Implicit* function definitions have empty function bodies but, instead, the properties of required result are expressed in a *postcondition* which has the form shown in Listing 5.6, while *explicit* function definitions have function bodies as indicated in Listing 5.7 [9].

```
1 function name (Parameters) result name and type
2 pre predicate
3 post predicate
```

Listing 5.6: The Form of Implicit Function Definitions

```
1 function name: Parameter types -> result types
2 function name (Parameters) ==
3     expression
4 pre predicate
5 post predicate
```

Listing 5.7: The Form of Explicit Function Definitions

In contrast, in Java, algorithms are defined by methods. Java methods also take parameters and produce results, and they may read and modify instance variables as VDM++ operations do. Not like VDM++ functions and operations, Java methods only have the explicit forms meaning that they should be defined explicitly. Furthermore, Java methods do not have *preconditions* and *postconditions.* In addition, like the VDM++ functions that are specified by "**is subclass responsibility**", Java methods can be abstract indicating that the method body must be implemented by any subclasses. It could be noticed that, in VDM++ specification, both implicit functions and functions that are specified by "**is not yet specified**" are intended to be left to the users or developers with necessary information (which is not formally specified) to complete, thus it is reasonable to leave these part to users by code generator as well, except that the code generator will generate "reminder" methods, which throws run-time error when called, reminding the user to complete the method, instead.

Rule 18: VDM++ implicit function definitions or functions that are specified by the keyword **`is not yet specified`** are mapped as Java methods which give specific error messages when called.

Listing 5.8 gives an example of a VDM++ class which contains implicit function and a function that is specified by "**`is not yet specified`**" and Listing 5.9 is the generated Java version of it.

```
1 class A
2 functions
3 public sqrt (x:rat) r: real
4 pre x>= 0
5 post r * r = x;
6 public f: int -> int
7 f(x) == is not yet specified;
8 end A
```

Listing 5.8: Implicit Function Definition

```
1 public class T {
2 public double sqrt (double x) {
3  throw new
4        InternalError("Method 'sqrt' is only defined implicitly");
5  }
6 public int f(int x) {
7  throw new InternalError("Method 'f' is not yet specified");
8  }
9 }
```

Listing 5.9: Generated Java Code for Listing 5.8

On the other hand, explicit function definitions are almost the same as Java methods, except that Java methods do not have preconditions and post conditions. Preconditions can be mapped as a Java if-then statement, but postconditions are a bit complicated to map and implement and they are not very necessary in implementation, postconditions are better to be ignored.

Rule 19: VDM++ explicit function definitions (excluding functions that are specified by the keyword '**`is not yet specified`**') are mapped as Java methods; VDM++ preconditions are mapped as Java conditional statements (such as Java if-then statement) which will be executed at the start of the methods to verify correct arguments and state; Postconditions are to be ignored while code generation.

By exploiting *rule 19,* Listing 5.10 will be generated to Java code as Listing 5.11

```
1 class A
```

```
functions
public foo: int * int -> int
      add(i,j) ==
                    (i/j)*(i/j)
pre j <>0
post RESULT >=0;
public Temp: int -> real
    Temp(n) == is subclass responsibility;
end A
```

Listing 5.10: Explicit Function Definition

```
public abstract class A {
  public int foo(int i, int j) {
     if(j==0)
       throw new InternalError("Precondition not satisfied");
     return (i/j)*(i/j);}
  public abstract double Temp(int n);
}
```

Listing 5.11: Java Translation of Listing 5.10

5.3.2 Operation Definitions

Operation definitions are almost the same as function definitions, except that operation definitions have operation body which can be a *statement* or can be specified by the keyword "**is subclass responsibility**" or "**is not yet specified**". Like function definitions, if an operation definition does not have an *operation body* , it is called implicit operation definition, otherwise it is categorized as explicit operation definition. Mapping of implicit and explicit operations to Java is the same as the mapping of implicit and explicit functions to Java, so the details regarding the mapping strategy of VMD++ operations are omitted here. Listing 5.12 shows an example of VDM++ operation definitions and Listing 5.13 demonstrates the generated Java code of Listing 5.12.

```
class D
instance variables
public i: int := 5;
Operations
public foo: int ==> int
      add(j) ==
                    return i + j
pre j >0
post RESULT >=0;
public Temp: int ==> real
    Temp(n) == is subclass responsibility;
end D
```

Listing 5.12: Examples of Operation Definitions

```
1 public abstract class D {
2   public int i = 5;
3   public int foo(int i, int j) {
4      if(j<=0)
5        throw new InternalError("Precondition not satisfied");
6      return (i+j);}
7   public abstract double Temp(int n);
8 }
```

Listing 5.13: Translation of Listing 5.12 to Java

5.4. Expressions

In this section, the coverage of mapping on VDM++ expressions are presented with necessary details. Detailed description about VDM++ expressions can be found in [5].

5.4.1 Unary and Binary Expressions

Table 5.1 and table 5.2 show how the VDM++ unary and binary expressions are mapped to Java. Some of the VDM++ expressions have corresponding Java constructs, so they are mapped to the methods in Java standard library, others (which do not have Java counterparts) are mapped to the methods that are defined in external library (see section 3.4.1 and section 6.4). CGCollections is one of the classes in code generator utility library (an external library, see section 6.4) which includes methods that are mentioned in the following tables.

VDM++ expressions	VDM++ operators	Java expressions	Library information
Equality	x = y	x.equals(y)	Java standard library
Inequality	x <> y	!x.equals(y)	Java standard library
Implication	x => y	Utils.implication(x,y)	External library
Biimplication	x <=> y	Utils.biimplication(x,y)	External library
Conjunction	x and y	x && y	Java standard library
Disconjunction	x or y	x || y	Java standard library
Sum	x + y	x + y	Java standard library
Subtract	x - y	x - y	Java standard library
Product	x * y	x * y	Java standard library
Division	x / y	x / y	Java standard library
Integer division	x div y	Utils.div(x, y)	External library
Remainder	x rem y	Utils.rem(x, y)	External library
Modulus	x mod y	Utils.mod(x, y)	External library
Power	x ** y	Math.pow(x, y)	Java standard library
Less than	x < y	x < y	Java standard library
Greater than	x > y	x > y	Java standard library
Less or equal	x <= y	x <= y	Java standard library
Greater or equal	x >= y	x >= y	Java standard library
In set expression	x in set y	y.contains(x)	Java standard library
Not in set expression	x not in set y	!y.contains(x)	Java standard library
Subset expression	x subset y	y.containsAll(x)	Java standard library
Proper subset	x psubset y	CGCollections.psubset(x, y)	External library
Set intersection	x inter y	CGCollections.inter(x, y)	External library
Set union	x union y	CGCollections.union(x, y)	External library
Set difference	x \ y	CGCollections.difference(x, y)	External library
Sequence concatenation	x ^ y	CGCollections.concatenation(x, y)	External library
Sequence Modification	x ++ y	CGCollections.SeqModification(x, y)	External library
Sequence Application	s(i)	s.elementAt(i - 1)	Java standard library
Map merge	x munion y	CGCollections.munion(x, y)	External library
Map override	x ++ y	CGCollections.Override(x, y)	External library
Domain restrict to	x <: y	CGCollections.DomTo(x, y)	External library
Domain restrict by	x <-: y	CGCollections.DomBy(x, y)	External library
Range restrict to	x :> y	CGCollections.RangeTo(x, y)	External library
Range restrict by	x :-> y	CGCollections.RangeBy(x, y)	External library
Map Iteration	x ** y	CGCollections.Iteration(x, y)	External library
Map composition	x comp y	CGCollections.comp(x, y)	External library
Map application	x (y)	x.get(y)	Java standard library

Table 5.1: Mapping of Binary Expressions

VDM++ Expressions	VDM++ Operators	Java Expressions	Java Library information
Unary Plus	+ x	+ x	Java standard library
Unary Minus	- x	- x	Java standard library
Absolute	abs x	Math.abs(x)	Java standard library
Floor	floor x	Math.floor(x)	Java standard library
Not	not x	!x	Java standard library
Set Cardinality	card x	x.size()	Java standard library
Power Set	power x	CGCollections.power(x)	External Library
Distributed Union Set	dunion x	CGCollections.dunion(x)	External Library
Distributed Union Set	dinter x	CGCollections.dinter(x)	External Library
(Sequence) Head	hd x	x.firstElement()	Java standard library
(Sequence) Tail	tl x	x.subList(1, x.size())	Java standard library
(Sequence) Length	len x	x.size()	Java standard library
(Sequence) Elements	elems x	CGCollections.elems(x)	External Library
(Sequence) Indexes	inds x	CGCollections.indexes(x)	External Library
Distributed Concatenation	conc x	CGCollections.conc(x)	External Library
Map Domain	dom x	x.KeySet()	Java standard library
Map Range	rng x	CGCollections.range(x)	External Library
Map Distributed Merge	merge x	CGCollections.merge(x)	External Library
Map inverse	inverse x	x.inverse()	External Library

Table 5.2: Mapping of Unary Expressions

5.4.2 Set Expressions

In VDM++, a set can be constructed using *set enumeration, set comprehension* and *set range expression*. Listing 5.14 contains examples of these three expressions.

```
1 class S
2 instance variables
3 --set enumeration
4 public Set1:set of nat := {1,2,3,12,34};
5 --set range expression
6 public Set2:set of nat := {1,...,4};
7 --set comprehension
8 public Set3:set of nat := {a | a in set Set1 & a in set Set2};
9 end S
```

Listing 5.14: Set Expressions

However, there are no similar constructs in Java, so three Java methods are defined in class `CGCollections.java` in code generator utilities package (see Appendix D) to provide Java constructs which do the same work as those three VDM++ set expressions. And the code generator only supports *set bind* and *in (or not in) set expression* (in predicate) in *set comprehension*.
Listing 5.14 will be generated to Java as Listing 5.15.

```
1 import org.overturetool.VDM2JavaCG.Utilities.*;
```

```
2  import java.util.HashSet;
3
4  public class S {
5  public  HashSet<Integer> Set1 =
6                   CGCollections.SetEnumeration(1,2,3,12,34);
7  public  HashSet<Integer> Set2 =
8                   CGCollections.SetRange(1,4);
9  public  HashSet<Integer> Set3 =
10                  CGCollections.SetComprehension(Set2,Set1);
11 }
```

Listing 5.15: Generated Java Code of Set Expressions

5.4.3 Sequence Expressions

The VDM++ sequence type can be constructed by three expressions, respectively *sequence enumeration, sequence comprehension* and *subsequence expression.* The sequence type is mapped as Vector type to Java (see section 4.2.2.2), and among the methods that available for Vector type in Java standard library, there is a method callled *sublist* which performs the same task as VDM++ *subsequence expression*. But the *sequence enumeration* and *sequence comprehension* don't have counterparts in Java standard library, so two methods, which performs same task on Java Vector type, are defined in class `CGCollections.java` of code generator utilities package (see Appendix D) to map these two VDM++ expressions to Java. Listing 5.16 shows examples of Sequence expressions and Listing 5.17 is the generated Java code of Listing 5.16.

```
1  class SQ
2  instance variables
3  --sequence enumeration
4  public Seq1:seq of nat := [1,2,3,12,34];
5  --sequence subset expression
6  public Seq2:seq of nat := Seq1(1, ..., 4);
7  --sequence comprehension
8  public Seq3:seq of nat :=
9               [a | a in set inds Seq1 & a in set elems Seq2];
10 end SQ
```

Listing 5.16: Sequence Expressions

```
1  import org.overturetool.VDM2JavaCG.Utilities.*;
2  import java.util.Vector;
3
4  public class SQ {
5  public  Vector<Integer> Seq1 =
6       CGCollections.SeqEnumeration(1,2,3,12,34);
7  public  Vector<Integer> Seq2 =
8       CGCollections.SubSequence(Seq1,0,3);
9  public  Vector<Integer> Seq3 =
```

```
10        CGCollections.SeqComprehension(CGCollections.indexes(Seq2),
11                                       CGCollections.elems(Seq1));
12  }
```

Listing 5.17: Generated Java Code of Sequence Expressions

In addition, currently the code generator only supports *set bind* and *in (or not in) set expression* (in predicate) in *sequence comprehension*.

5.4.4 Map Expressions

A VDM++ map can be constructed by *map enumeration* or *map comprehension. Map enumeration* expression consists of a set of *maplet* expressions. *Maplet expression* has the following form:

```
1  expression |-> expression ;
```

Listing 5.18: Maplet Expression

Maplet expression and *Map enumeration* are mapped as Java methods `MapLet` and `MapEnumeration` which are defined in class `CGCollections.java` of code generator utilities package (see Appendix D). Listing 5.19 shows an example of *map enumeration* and it is translated to Java as shown in Listing 5.20.

```
1  class M
2  instance variables
3  --Map enumeration
4  public Map1: map nat to seq of char := {1 |-> "one", 2|-> "two"};
5  end M
```

Listing 5.19: Map Enumeration

```
1  import org.overturetool.VDM2JavaCG.Utilities.*;
2  import java.util.HashMap;
3
4  public class M {
5  public  Map<Integer, String> Map1 =
6      CGCollections.MapEnumeration(CGCollections.MapLet(1, "one"),
7                                   CGCollections.MapLet(2, "two"));
8  }
```

Listing 5.20: Generated Java Code of Sequence Expressions

Due to the complexity of the mapping of *map comprehension* to Java, currently *map comprehension* is not supported by the code generator.

5.4.5 Conditional Expressions

The VDM++ conditional expressions - *if expressions* and *cases expressions* make choice of one from a set of expressions in terms of the value of a particular expression [5].
The *if expression* has the following form:

```
1 if e1
2 then e2
3 else e3
```

Listing 5.21: The VDM++ If Expression

where `e1` is a boolean expression, `e2` and `e3` can be expressions of any type. If `e1` evaluates to true the *if expression* returns the value of `e2` evaluated in the given context, otherwise it returns the evaluated value of `e3`. In Java, there is no *if expression*, but there is *if statement* which has the following form:

```
1 if (e)
2     s1;
3 else
4    s2;
```

Listing 5.22: The Java If Statement

where `e` is an boolean expression, `s1` and `s2` are statements. If `e` evaluates to true, then s1 is executed, otherwise s2 is executed. Since, in VDM++ an *if expression* returns the value of one of the evaluated alternative expressions, the alternative expressions of a VDM++ *if expression* can be mapped as Java *return statement.* So the Listing 5.21 is translated to Java as Listing 5.23.

```
1 if (e1)
2    return e2;
3 else
4    return e3;
```

Listing 5.23: Generated of Java Code of Listing 5.21

Mapping of the VDM++ *elseif expression* to Java is the same as the *if expression.*
The *cases expression* has the following form:

```
1 cases e:
2 p11, p12, ..., p1n -> e1,
3 ...                -> ...,
4 pm1, pm2, ..., pmk -> em,
5 others             -> emplus1
6 end
```

Listing 5.24: The VDM++ Cases Statement

where e is an expression of any type, all pij's are patterns which are matched one by one against the expression e and the e's are expressions of any type [5]. The *cases expression* returns the value of the ei expression evaluated in the context in which one of the pij patterns has been matched against e. The Java *switch statement* is similar to VDM++ *cases expression,* however, in Java *switch statement* the patterns have to be constants [15], while in VDM++ *cases expression* the patterns do not have to be constants [5]. So it is not possible to map the *cases expressions* to Java, hence they are to be ignored while code generating.

5.4.6 Record Expressions

The VDM++ record expressions are *record constructor* and *record modifier*. The *record type* is mapped as a Java class and the fields of the *record type* are mapped as instance variables in the Java class (see section 4.2.2.5). In addition, the *record constructor* is mapped as a constructor of Java class, which is presented in section 4.2.2.5 in detail.
The *record modifier* is used to modify the fields of a value of record type which has the form shown in Listing 5.25:

```
1 mu (e, id1 |-> e1, id2 |-> e2, ..., idn |-> en)
```

Listing 5.25: The Record Modifier

where e is the record value to be modified and all identifiers idi must be distinct names of the fields in the record type of e. Mapping of such expression can be done in a simple way which is to map the expression to Java as a block of Java *assignment statements* which modify the instance variables of the corresponding object. Consequently, Listing 5.25 can be translated to Java as shown in Listing 5.26.

```
1 { e.id1 = e1;
2   e.id2 = e2;
3   ...
4   e.idn = en;
5 }
```

Listing 5.26: Generated Java Code

However, currently this strategy only works for some cases.

5.4.7 The New Expression, Self Expression and Is Expression

The VDM++ *new expression* is mapped as Java *new expression.* The VDM++ *self expression* is mapped to Java as Java *this expression* which is specified by the keyword '**this**'.
Finally, the VDM++ *is expression* is mapped to Java as *instaceof expression,* which is specified by the keyword '**instanceof**' (see section 3.2.8 for details), in appropriate ways.

5.4.8 Class Memberships

Since every class in Java has the same root class which is java.long.Object, so VDM++ *base class membership* and *same base class membership* expressions are no supported by the code generator.

The *class membership* expression is mapped to Java as *instanceof expression* in an appropriate way. And the *same class membership* expression is mapped to Java as an *equals expression* in appropriate way. For instance, the expression in Listing 5.27 is translated to Java as shown in Listing 5.28.

```
1 sameclass(b,t)
```

Listing 5.27: Same Class Membership

```
1 b.getClass().equals(t.getClass());
```

Listing 5.28: Generated Java Code

5.4.9 The Undefined Expression

The VDM++ *undefined expression* is mapped as a Java *null expression* in Listing 5.29:

```
1 null
```

Listing 5.29: The Throw Statement

5.5. Statements

Statements are used in the bodies of explicit operations to describe and algorithm, typically affecting the instance variables in a class [9]. This section presents the mapping strategies of VDM++ statements to Java. Descriptions about the VDM++ statements can be found in VDM++ [9] and VDM++ language manual [5].

5.5.1 The Block Statement

The block statement enables the use of locally defined variables that can be modified inside the body (between '(' and ')') of block statement denoting the ordered execution of the each individual statement inside the body [5]. The form of the block statement is shown in Listing 5.30.

```
1  class B
2  instance variables
3  public x:nat :=1;
4  public y:nat :=2;
5  operations
6  public Swap: () ==> ()
7         Swap () ==
8          (dcl temp: nat := x;
9           x := y;
10          y := temp)
```

```
11 end B
```

Listing 5.30: The VDM++ Block Statement

Fortunately, there is a very similar also called block statement in Java and the body of it is enclosed between '{' and '}'. The Java block statement can include local variable declaration statement, class declaration and any other statements and the statements inside the body are also executed in order (from first to last) [15]. Hence, the VDM++ block statement can be mapped as the Java block statement.

Rule 20: The VDM++ block statements are mapped as Java block statements.

So, VDM++ specification in Listing 5.30 is translated to Java as shown in Listing 5.31.

```
1
2  public class B {
3  public int x = 1;
4  public int y = 2;
5  public void Swap() {
6     {int temp = x;
7           x = y;
8           y = temp;}
9  }
10
11 }
```

Listing 5.31: The Java block statement

5.5.2 The Assignment Statement

The VDM++ assignment statement is used to modify the value of the global or local state, likewise the Java assignment statement is used to change the value of the global or local state. An assignment statement can be a single *assign statement* or *multiple assign statements,* So the VDM++ *assign statement* can be mapped as the Java assignment statement and the *multiple assign statements* can be mapped as a set of Java statements that are enclosed between '{' and '}'. For instance, Listing 5.32 shows the two different forms of the assignment statement and Listing 5.33 displays the Java translation of Listing 5.32.

```
1 --assign statement
2 sd := ec;
3 --multiple assign statement
4 atomic (sd1 := ec1;
5         ...;
6         sdN := ecN)
```

Listing 5.32: The VDM++ Assignment Statement

```
sd = ec;
{ sd1 = ec1;
  ...;
  sdN = ecN;}
```

Listing 5.33: Java Assignment Statements

5.5.3 Conditional Statements

There are 2 conditional statements in VDM++, respectively *if statement* and *cases statement*. The VDM++ *if statement* can be easily mapped as Java *if statement,* since they are almost the same. However, the VDM++ *cases statement* can not be mapped to Java straightforwardly. The semantics of *cases statement* corresponds to the *cases expression* (see section 5.4.5) except for the alternatives which are statements [5]. As shown in Listing 5.34, the alternatives of *case statement* are statements which are s1, ..., sm, smplus1.

```
cases e:
p11, p12, ..., p1n -> s1,
...                -> ...,
pm1, pm2, ..., pmk -> sm,
others             -> smplus1
end
```

Listing 5.34: The VDM++ Cases Statement

So, like the *case expression* (see section 5.4.5), the *case statement* cannot be mapped to Java for the same reason.

5.5.4 For-Loop Statements

The VDM++ *for-loop statements* are all mapped as Java *for statements* in appropriate ways.

5.5.5 The While-Loop Statement

The VDM++ *while-loop* statement is mapped as Java *while statement*.

5.5.6 The Call Statement

The VDM++ *call statement* is mapped as Java *method invocation expression statement* (see [15] for detail).

5.5.7 The Return Statement

The VDM++ *return statement* is mapped as Java *return statement.*

5.5.8 The Error Statement

The VDM++ *error statement* is mapped as a Java *throw statement* which throws a 'The result is undefined' error which following form:

```
1 throw new
2       InternalError("The result of the statement is undefined");
```

Listing 5.35: The Throw Statement

5.5.9 The Identity Statement

The VDM++ *identity statement* which is specified by the keyword **skip** is mapped as Java *continue statement* (see [15] for more detail) which is specified with the keyword **continue** with no label.

Chapter 6

Development Components and Structures

In this chapter, on the basis of the transformation rules and strategies presented in chapter 4 and chapter 5, abstract syntax tree (AST) transformation from VDM++ to Java and the final process of generating Java source file (Back-end) is illustrated.

6.1. Abstract Syntax Tree

This section mainly discusses the abstract syntax tree of Java and VDM++, and the transformation from Java to VDM++.

6.1.1 Definition of AST

An abstract syntax tree (AST), is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code. The syntax is 'abstract' in the sense that it does not represent every detail that appears in the concrete syntax. For instance, grouping parentheses are implicit in the tree structure, and a syntactic construct such as an if-then-else expression may be denoted by a single node with two branches [22]. In other words, AST of a block of source code written in certain language is a view or presentation of that block of source code in the form of tree structure based on that programming language's syntatic structure and rules. For instance, figure 6.1 show abstract syntax tree of if-then-else statement in the method `foo` inside a Java class in Listing 6.1.

```
1 public class T {
2    public int Foo(int x, int y) {
3         if(x > y)
4           return x;
5         else
6           return x+y;
7             }
8 }
```

Listing 6.1: A Java Class

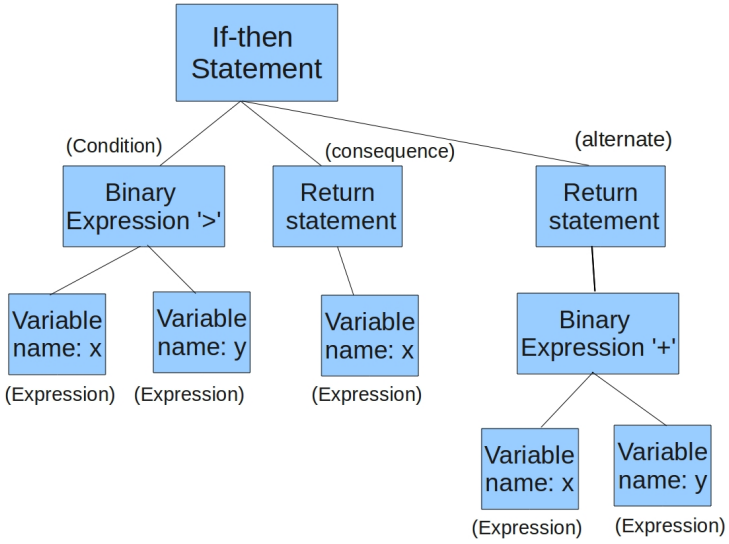

Figure 6.1: AST of If-then-else Statement

6.1.2 VDMJ AST

The VDMJ AST is implemented in the Overture VDMJ tool which provides tool support for the VDM-SL, VDM++ and VDM-RT specification languages, written in Java [1]. The tool consists of a parser, a type checker, an interpreter, a debugger and a proof obligation generator (see the design specification [1] for detail). The code generator uses VDMJ parser to extract the AST of VDM++ source code. The VDMJ AST specification is attached in the appendix B and detailed documentation can be found in VDMJ design specification [1]. As VDMJ is written in Java, nodes of the VDMJ AST is implemented as a Java class (or abstract class) and the parent-child relations between nodes are represented by Java class hierarchy. For instance, class Expression is the super class of all other specific expressions, such as Set Expression, Sequence concatenation Expression, etc, in further class SetExpression is the super class or parent node of set enumeration, set range expression, set comprehension nodes.

6.1.3 Java AST and Overture TreeGen

This section introduces Java AST and Overture TreeGen, and the relation between them.

6.1.3.1 Java AST

A Java AST specification is defined in VDM++ based on comparison with VDMJ AST for the sake of covenience and simpilicity of the AST transformation from VDM++ to Java. The full Java AST specification is attached in appendix C. As shown in Listing 6.2 , specification is the root node of the Java AST which consists of class definitions (comparable to the root node 'Definition List' of the VDMJ AST) and interface definitions.

```
Specification =  ClassDefinitions |
                 InterfaceDefinitions;
ClassDefinitions ::
                   class_list: seq of ClassDefinition;
InterfaceDefinitions ::
                     interface_list: seq of InterfaceDefinition;

ClassDefinition ::
                 accessdefinition : AccessDefinition
                 modifiers         : Modifier
                 identifier : Identifier
                 inheritance_clause : [InheritanceClause]
                 body : DefinitionList;

InterfaceDefinition ::
                     accessdefinition : AccessDefinition
                     modifiers         : Modifier
                     identifier : Identifier
                     inheritance_clause : [InheritanceClause]
                     body : DefinitionList;

InheritanceClause ::
                    identifier_list : seq of Identifier;
                    Modifier ::
                    Abstract : bool
                    Final    : bool ;
DefinitionList ::
                 definition_list : seq of Definition;

Definition =    InstanceVariableDefinition |
                 MethodDefinition |
                 ClassDefinition |
                 InterfaceDefinition;
```

Listing 6.2: Java AST

6.1.3.2 Overture TreeGen

The Overture TreeGen is a feature of Overture IDE (see [20]) that is used to generate Java code to manage abstract syntax trees which are defined as type definitions in VDM++.

6.2. AST Transformation

This section specifies AST transformation between VDM++ and Java. The VDMJ parser and type-checker parses and then type checks the VDM++ source file and saves the VDM++ values

with type information in an AST which is at VDM++ level. The class `Vdm2Java.java` has formulated the transformation rules that are specified in chapter 4 and chapter 5. `Vdm2Java.java` walks through the AST of VDM++ file mapping the VDM++ constructs in each node of the AST exploiting the transformation rules that are defined in the thesis. And then it stores the translated Java constructs in an AST at Java level (see section 6.1.3).

6.2.1 Overview

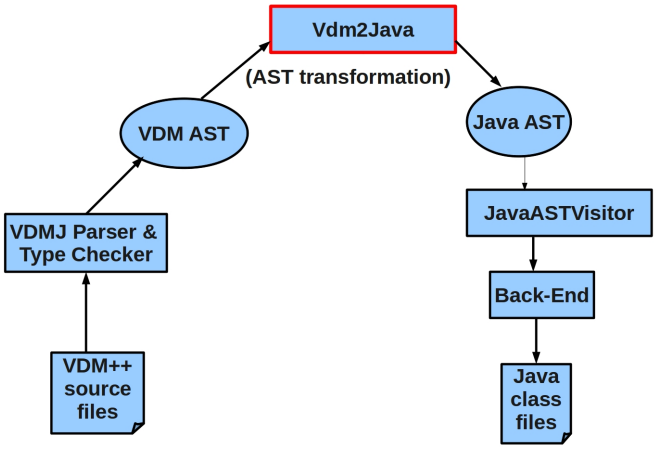

Figure 6.2: Overview

6.3. Back-end

The back-end consists of two Java classes, respectively `JavaASTVisitor.java` and `BackEnd.java`. They take care of the final process of code generation.

6.3.1 Java AST Visitor

The class `JavaASTVisitor.java` walks down the AST in Java level that is produced by the class `vdm2java.java` and then produces correct code conforming to the syntax and semantics of Java.

6.3.2 Printing Out the Resulting Java Source File

The class `BackEnd.java` takes care of printing out the Java code produced by the class `JavaASTVisitor.java` as Java source file.

6.4. Code Generator Utilities Library

The code generator utilities library (package `org.overturetool.VDM2JavaCG.Utilities`) includes rich Java methods and other classes that are not included in the standard library but essential to compile and execute the generated Java code (for details see section 3.4.1 on page 14

and Appendix D).

Chapter 7

AST Transformation Challenges and Limitations

This chapter presents the challenges during this work and the limitation of the code generator.

7.1. Union Type

There is no similar construct in Java because of the concern regarding the type-safety issue of the Java code, hence the mapping of the union type is a major challenge, especially it is hard to ensure the type-safety of the generated Java code from VDM++ union types. So the VDM++ union type is restricted to one form (which is presented in section 4.2.2.6 on page 23 in details).

7.2. Type Definitions

Java type system consists of primitive types and the classes (class types) defined in the Java standard library, defining a new type is not possible in Java. So the translation of type definitions is another challenge in this thesis. But it is possible to define a class serving for the intended purpose, though this loses some of the advantages of the type.

7.3. Invariants

In Java there is no invariants, so the invariants in the VDM++ specification is omitted in the generated Java code.

7.4. Implicit Functions/Operations

In Java methods could only be defined in explicit way, so it is no possible to generate Java methods for the VDM++ implicit functions and operations, instead it is possible to generate skeleton Java code for them (see section 5.3 on page 32 for details).

7.5. Pre & Post Conditions in Functions/Operations

Since there is no pre or post conditions in Java methods, it becomes difficult to map the pre and post conditions in VDM++ functions and operations. The pre condition has been dealt in an appropriate way (see section 5.3 on page 32), however the post condition is ignored by the code generator.

7.6. Expressions and Statements

Following is the list of expressions and statements that are not supported or partially supported by the code generator largely because of the significant differences between the VDM++ and Java.

Types:

Name	level
Product type	not supported
Function type	not supported
Union type	partially support

Expressions:

Name	level
Let expression	not supported
Define expression	not supported
Iota expression	not supported
Tuple constructor expression	not supported
Threadid expression	not supported
Lambda expression	not supported
History expression	not supported
Cases expression	not supported
quantified expression	not supported

Statements:

Name	level
Let statement	not supported
Define statement	not supported
Start and start list statement	not supported
Error statement	not supported
Cases statement	not supported

Let statement, define statement, define expression and let expression are not supported because the VDMJ AST does not provide type information for the names in those expressions and statements.

7.7. Multiple Inheritance

In Java one class could only have one super class whereas in VDM++ it is possible for one class to inherit from multiple super classes, so the multiple inheritance is no supported by the code generator. The code generator gives error messages when A VDM++ specification including multiple inheritance (or other no supported constructs).

Chapter 8

Concluding Remarks and Future Work

8.1. Summary

To achieve the thesis objective introduced in section 1.1 on page 3, a thorough study on VDM++ and Java has been carried out. Syntax and semantics of VDM++ and Java have been analyzed. Based on the analysis result, mapping possibilities from VDM++ constructs to Java constructs are investigated, subsequently a number of transformation rules are determined and formulated in Java. During the process, knowledge about the type-safety of the generated code is explored and the type system of VDM++ and Java further studied in order to map the VDM++ constructs to Java type-safely. In order to implement the transformation rules and automate the code generation from a VDM++ class file to a Java class file, a Java class (the class `vdm2java.java` in the source code, which is a component of the code generator) is built to specify a transformation from VDM++ level to Java level. The AST for VDM++ is specified in VDMJ AST which existed before starting this thesis and available as a part of Overture project (see section 6.1.2 for detail). The Java AST is inspired by the VDMJ AST and specified as type definitions in VDM++ and then generated to Java interfaces and implementations (of the generated interfaces) using the Overture tool Treegen (see section 6.1.3). During those process, the VDMJ AST, parser and type-checker are studied in order to get a better understanding of the AST structure of VDM++. And the Overture tool Treegen is also explored in providing an AST for the generated Java.
The VDM++ value and type information are extracted from VDM++ source file using the VDMJ parser and type-checker and are stored in an AST (in VDM++ level). In the next step, each VDM++ construct is visited and mapped (by an object of the class `vdm2java.java` in source code) to Java constructs by walking through the AST of VDM++ file, subsequently the resulting Java constructs are stored in an AST at the Java level. A back-end (class `JavaASTVisitor.java` and `BackEnd.java` in the source code) is developed in order to walk down the generated Java AST and generate the corresponding Java source file. The generated Java source file has been tested in terms of type-safety and to see if it is semantically equivalent to VDM++ specification and also to check if it produces the same result as the VDM++ model does.

8.2. Assessment of the Thesis Work

The main purpose of this thesis is to explore the mapping potentials from VDM++ to Java to determine to which degree it is possible to generate type-safe Java code from VDM++ specification and then implement the ideas in a Java application on the Overture platform to verify the theories (mapping rules) stated in the thesis and found the basis for a future tool feature which is able to

generate type-safe Java code from VDM++ specification in an optimal way. As a result of this thesis, a large subset of VDM++ constructs have been mapped to Java in a type-safe way. A small subset of VDM++ constructs that cannot be mapped to Java have been identified and delt in an appropriate fashion in generated Java code. In the meanwhile, a code generation tool has been developed and well documented in the thesis.
In addition, comparing to the existing tools (especially the VDMTools Java code generator), a main break through in this thesis is the restrictions on the usage of union types in VDM++ specification to ensure the type-safety in the generated Java code, details on the mapping and limitation of the union type is presented in section 4.2.2.6 on page 23. In addition, thanks to the generics deployed in the current Java version (Java SE 6), the mapping of VDM++ collections to Java collections became convenient and type-safety in the generated Java code is improved to a large degree comparing to the VDMTools (for more details see section 2.3 on page 8). And also with the help of rich utilities offered by the Google collections library in Google Guava project (see section 3.4.1 on page 14), the mapping of operations and expressions on VDM++ collections to Java became simple and convenient.
In this regard, it can be concluded that the main objective of this thesis (see section 1.3 on page 2) has been achieved, though there are spaces to enrich this work and extend the tool developed in this thesis. There are a small subset of VDM++ constructs which are not supported or partially supported (see chapter 7) by the code generator due to different reasons. For instance, *let expression* is not supported since the type information of the name in the expression can not be obtained due to the limitation of the VDMJ AST (see section 7.6 for details), but it is expected to be fixed by the realese of a new AST (my colleagues are working on this currently) in the near future. Restricting the union type to be only formed by record or quote types has improved the type-safety of the generated Java code, but the mapping of union type to Java can be studied in wider range to reduce some of the restrictions. In addition, though the limitations and challenges (such as unsupported VDM++ constructs) of the code generator has been documented (in chapter 7), it still can be formalized into the syntax and semantics of VDM++ to develop a implementation-oriented version of VDM++ which could be fully covered by the code generator. Finally, although this thesis has specified a code generator which is able to generate type-safe Java code from VDM++ constructs, but there are very little investigation regarding the user-interaction with the code generator in a user-friendly way especially when making mapping decisions for some VDM++ constructs which need user intervention (such as more details from the user).

8.3. Future Work

Concerning the future work, the limitations (see chapter 7 on page 52) and the challenges stated in chapter 7 can be further studied. A more user friendly and interactive graphical interface which gives the user the option of making mapping decision for some constructs that has alternative mapping strategies could be developed. Finally the mapping possibilities of union type to Java could be further studied to potentially loosen up some of the restrictions on the usage of union type to keep its advantage of flexibility in VDM++ specifications.

8.4. Concluding Remarks

This thesis presented the design and development of a Java code generator for VDM++. The tool takes a model specification written in VDM++ and generates compilable and type-safe Java code covering a large amount of VDM++ constructs. Even though the tool is built to generate Java code, it would not be so hard to make it generate C++ code or any other object-oriented programming language code, but it also depends on the challenges in semantic differences between those languages and Java. In addition, this thesis has specified the differences between VDM++ and Java constructs and the mapping possibilities and challenges from VDM++ to Java which hopefully will ease the future work of others on the same or related subject. I sincerely hope that, this work could ease the work of researchers in any field and pave a way for future developments in the area of code generators.

Bibliography

[1] Nick Battle. *VDMJ Design Specification Issue.1.1*, 10 2010. [cited at p. 8, 48]

[2] E. Bonta, M. Bernardo, and Ieee. *PADL2Java: A Java Code Generator for Process Algebraic Architectural Descriptions*. 2009 Joint Working Ieee/Ifip Conference on Software Architecture and European Conference on Software Architecture. Ieee, New York, 2009. ISI Document Delivery No.: BNK27 Times Cited: 0 Cited Reference Count: 26 Bonta, Edoardo Bernardo, Marco Proceedings Paper Joint Working IEEE/IFIP Conference on Software Architecture/European Conference on Software Architecture SEP 14-17, 2009 Cambridge, ENGLAND IEEE, IFIP 345 E 47TH ST, NEW YORK, NY 10017 USA. [cited at p. 3]

[3] CSK. VDM homepage. *http://www.csk.com/support_e/vdm/index.html*, 2005. [cited at p. 8]

[4] CSK. VDMTools homepage. *http://www.vdmtools.jp/en/*, 2007. [cited at p. 1, 8]

[5] CSK. *VDMTools: The VDM++ Language Manual Ver.1.0*. CSK Systems Corporation(Japan), 2009. http://www.csk.com/systems. [cited at p. 6, 19, 32, 33, 36, 41, 42, 43, 45]

[6] CSK. *VDMTools: The VDM++ to Java Code Generator Ver.1.1*. CSK Systems Corporation(Japan), 2010. http://www.csk.com/systems. [cited at p. 8, 9]

[7] John Fitzgerald, Cliff B. Jones, and Peter Lucas, editors. *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods (Proc. 4th Intl. Symposium of Formal Methods Europe, Graz, Austria, September 1997)*, volume 1313 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1997. ISBN 3-540-63533-5. [cited at p. 1]

[8] John Fitzgerald and Peter Gorm Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 1998. ISBN 0-521-62348-0. [cited at p. 6]

[9] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs for Object–oriented Systems*. Springer, New York, 2005. [cited at p. 1, 6, 7, 8, 9, 16, 21, 23, 32, 33, 43]

[10] John Fitzgerald, Peter Gorm Larsen, and Shin Sahara. VDMTools: Advances in Support for Formal Modeling in VDM. *ACM Sigplan Notices*, 43(2):3–11, February 2008. [cited at p. 8]

[11] Ed Felten Gary McGraw. *Securing Java*. John Wiley & Sons, Inc., 1999. [cited at p. 14]

[12] Ed Felten Gary McGraw. Securing Java. http://www.securingjava.com/chapter-two/chapter-two-10.html, 1999. [cited at p. 14]

[13] Google. Google Guava Libraries. http://code.google.com/p/guava-libraries/, 2011. [cited at p. 14]

[14] The VDM Tool Group. The IFAD VDM++ Language. Technical report, CSK Systems, January 2008.
ftp://ftp.ifad.dk /pub/vdmtools/doc/langmanpp_letter.pdf. [cited at p. 8, 19, 20, 22]

[15] Guy Steele Gilad Bracha James Gosling, Bill Joy. *The Java Language Specification Third Edition*. The Java Series. ADDISON-WESLEY, 2005. [cited at p. 12, 13, 42, 44, 45, 46]

[16] Cliff B. Jones. Scientific Decisions which Characterize VDM. In J.M. Wing, J.C.P. Woodcock, and J. Davies, editors, *FM'99 - Formal Methods*, pages 28–47. Springer-Verlag, 1999. Lecture Notes in Computer Science 1708. [cited at p. 6]

[17] Roberto Tamassia Michael T. Goodrich. *Data Structures and Algorithms in Java, Fifth edition*. John Wiley & Sons, Inc., Asia, 2011. [cited at p. 10, 11, 13]

[18] Oliver Oppitz. Concurrency extensions for the vdm++ to java code generator of the ifad vdm++ toolbox. Master's thesis, TU Graz, Austria, April 1999. [cited at p. 2]

[19] Oracle. Java SE 6 Documentation. http://download.oracle.com/javase/6/docs/, 2011. [cited at p. 11, 13]

[20] Overture-Core-Team. Overture Web site. http://www.overturetool.org, 2007. [cited at p. 1, 10, 49]

[21] T. Taibi and T. Mkadmi. Generating java code from design patterns formalized in bpsl. *2006 Innovations in Information Technology (IEEE Cat. No.06EX1543C)*, pages 5 pp.|CD–ROM, 2006. Times Cited: 0 2006 Innovations in Information Technology Dubai United Arab Emirates. [cited at p. 3]

[22] Wikipedia. Abstract Syntax Tree. http://en.wikipedia.org/wiki/Abstract_syntax_tree, 2011. [cited at p. 47]

[23] Wikipedia. Java Interface. http://en.wikipedia.org/wiki/Interface_(Java), 2011. [cited at p. 12]

[24] Wikipedia. Java primitive wrapper class. http://en.wikipedia.org/wiki/Primitive_wrapper_class, 2011. [cited at p. 11]

[25] Wikipedia. Java Programming language. http://en.wikipedia.org/wiki/Java_(programming_language), 2011. [cited at p. 11]

[26] Wikipedia. Type Safety. http://en.wikipedia.org/wiki/Type_safety#Type-safe_and_type-unsafe_languages, 2011. [cited at p. 14]

[27] Wikipedia. Vienna Development Method. http://en.wikipedia.org/wiki/Vienna_Development_Method, 2011. [cited at p. 1]

Appendices

Appendix A

Abbreviations

AST
: Abstract Syntax Tree

VDM
: Vienna Development Method

VDM-SL
: VDM Specification Language

VDM-RT
: VDM Real Time

IT
: Information Technology

BPSL
: Balanced Pattern Specification Language

PADL
: Process Algebraic Architectural Descriptions Language

Appendix B

VDMJ AST

VDMJ abstract syntax tree (AST) is included in CD.

Appendix C

Java AST

Following is the Java AST specified in VDM++ in such a way that the transformation from VDM++ AST to Java AST becomes convenient and easy.

```
--
-- OVERTURE Java ABSTRACT SYNTAX DEFINITION
--


class Java
values
  package = "org.overturetool.VDM2JavaCG.ast.java";
  directory = "/home/mehmudjan/workspace/vdmpp2javaCG/src";
  toplevel = "Specification";

types

Specification =
                                ClassDefinitions |
                        InterfaceDefinitions;

ClassDefinitions ::
        class_list: seq of ClassDefinition;

InterfaceDefinitions ::
        interface_list: seq of InterfaceDefinition;


----
---- Java CLASS DEFINITIONS
----

ClassDefinition ::
        Nested: bool
    accessdefinition : AccessDefinition
    modifiers         : Modifier
```

```
33          identifier : Identifier
34          inheritance_clause : [InheritanceClause]
35          body : DefinitionList;
36
37  InterfaceDefinition ::
38          accessdefinition : AccessDefinition
39      modifiers          : Modifier
40          identifier : Identifier
41          inheritance_clause : [InheritanceClause]
42          body : DefinitionList;
43
44
45  InheritanceClause ::
46          implement_list : seq of Identifier
47          extends: seq of Identifier;
48
49  Modifier ::
50       Abstract : bool
51       Final    : bool ;
52
53
54  ----
55  ---- CORE DEFINITIONS
56  ----
57
58  DefinitionList ::
59          definition_list : seq of Definition;
60
61  Definition =
62          InstanceVariableDefinition |
63  --      ConstructorDefinition |
64          MethodDefinition |
65          AssignmentDefinition |
66      ClassDefinition |
67          EmptyDefinition |
68      InterfaceDefinition;
69
70  EmptyDefinition::;
71
72  AssignmentDefinition::
73          identifier: Identifier
74          type: Type
75          expression: Expression;
76
77  AccessDefinition ::
78      static_accsess  : bool
79          scope  : Scope;
80
81  Scope =
```

```
        <PUBLIC> | <PRIVATE> | <PROTECTED>;

---
--- METHOD DEFINITIONS
---

MethodDefinition ::
        IsConstructor: bool
        access : AccessDefinition
    modifiers : Modifier
        identifier : Identifier
        returntype : Type
        parameter_list : seq of Bind
        body : MethodBody;

--ConstructorDefinition ::
        --access : AccessDefinition
    --modifiers : Modifier
        --identifier : Identifier
        --parameter_list : seq of Bind
        --body : MethodBody;

MethodBody ::
        precondition: [Expression]
        statement : [seq of Statement]
        subclass_responsibility : bool
        Implicit: bool
        postcondition: [seq of char];

---
--- InstanceVariable Definitions
---

InstanceVariableDefinition ::
                                                access : AccessDefinition
                                        modifiers : Modifier
                                        type : VariableType
                                                name: Identifier
                                                expression: Expression
                                                initialized: bool;
                --SimpleField |
        --InitializedField ;


--SimpleField ::
        --access : AccessDefinition
    --modifiers : Modifier
    --type : VariableType
    --name: Name;
```

```
131
132 --InitializedField ::
133         --access : AccessDefinition
134     --modifiers : Modifier
135     --type : VariableType
136     --name: Name
137         --expression: Expression
138         --initializer: [Initializer];
139
140 --Initializer = Static_Init |
141         --                          Init;
142
143 --Static_Init::
144                 --      body: Expression;
145
146 --Init::
147         --      body: Expression;
148
149 VariableType =
150                         Type |
151                         ReferenceType;
152
153 ReferenceType ::
154                 class_identifier: Identifier;
155
156
157
158 ---
159 ---  types
160 ---
161
162 Type =
163         ByteType |
164         ShortType |
165         IntType |
166         LongType |
167     FloatType |
168         DoubleType |
169         CharType |
170     StringType |
171     BooleanType |
172     VectorType |
173         ClassType |
174         VoidType |
175         MapType |
176         SetType |
177         ProductType |
178         TokenType |
179         QuoteType |
```

```
180         UnresolvedType;
181
182
183 ByteType  :: ;
184
185 ShortType :: ;
186
187 IntType :: ;
188
189 LongType :: ;
190
191 FloatType :: ;
192
193 DoubleType ::;
194
195 CharType :: ;
196
197 TokenType :: ;
198
199 ClassType::
200         name: seq of char;
201
202 QuoteType ::
203         Literral: seq of char;
204
205 VoidType::
206         isReturnType: bool;
207
208 UnresolvedType ::
209         typename: Identifier;
210
211 StringType ::
212         string : seq of char;
213 VectorType ::
214         type : Type;
215
216 MapType =
217         GenMap |
218         BiMap;
219
220 GenMap::
221                 key: Type
222                 value: Type;
223 BiMap::
224                 key: Type
225                 value: Type;
226 SetType::
227                 type: Type;
228 ProductType::
```

```
                Types: seq of Type;


BooleanType :: ;

-----
--- EXPRESSIONS
---

Expression =
        Name |
        ApplyExpression |
        CardinalityExpression |
        SymbolicLiteralExpression |
        StringLiteralExpression |
        QuoteLiteralExpression |
        VectorEnumExpression |
        VectorExpression |
        SeqCompExpression |
        VectorConcatExpression |
        ElementsExpression |
        SubVectorExpression |
        LengthExpression |
        IndexesExpression |
        DistConcat |
    IfExpression |
        ElseIfExpression |
    UnaryExpression |
        ImplicationExpression|
        BiimplicationExpression|
        IntDivExpression|
        RemainderExpression |
        ModulusExpression |
    BinaryExpression |
    EqualsExpression |
        EmptyExpression |
        HeadExpression |
        CompositionExpression |
        TailExpression |
        PlusPlusExpression |
        SetExpression |
        MapExpression |
        StarStarExpression |
        VectorApplication |
        NewExpression |
        ThisExpression |
        IsExpression |
        IsBasicTypeExpression |
        RecordExpression |
```

```
278         IsSubClassResponsibilityExpression |
279         NotYetSpecifiedExpression |
280         SameClassMembership|
281     TokenExpression |
282         TupleConsExpression;
283
284
285 TokenExpression::
286         Value: Expression;
287
288 QuoteLiteralExpression::
289         Literal: seq of char;
290
291 SameClassMembership::
292         left: Expression
293         right: Expression;
294
295 IsExpression::
296         ClassName: Expression
297         InstanceName: Expression;
298
299 IsBasicTypeExpression::
300         type: Type
301         Value: Expression;
302
303 NewExpression::
304         ClassName: Expression
305         expressions: seq of Expression;
306
307 ThisExpression::;
308
309 ImplicationExpression::
310         left: Expression
311         right: Expression;
312
313 BiimplicationExpression::
314         left: Expression
315         right: Expression;
316
317 IntDivExpression::
318         left: Expression
319         right: Expression;
320
321 RemainderExpression::
322         left: Expression
323         right: Expression;
324 ModulusExpression::
325         left: Expression
326         right: Expression;
```

```
327
328 IsSubClassResponsibilityExpression ::;
329
330 NotYetSpecifiedExpression ::;
331
332 CompositionExpression::
333         left: Expression
334         right: Expression;
335
336 EmptyExpression::;
337
338 Name ::
339         identifier : Identifier;
340
341 ApplyExpression ::
342         expression : Expression
343         expression_list : seq of Expression;
344
345 CardinalityExpression::
346                 name: Expression;
347
348 SymbolicLiteralExpression ::
349         literal : Literal;
350
351 StringLiteralExpression ::
352         string: seq of char;
353
354 TupleConsExpression::
355         owner: Identifier
356         args: seq of Expression;
357
358 VectorEnumExpression ::
359         elements : seq of Expression;
360
361 VectorExpression ::
362         genericType: Type
363         elements: seq of Expression;
364 VectorApplication ::
365                 Name: Expression
366                 at: Expression;
367
368 SeqCompExpression::
369                 bindlist: seq of Bind
370                 predicate: [Expression];
371
372 VectorConcatExpression ::
373         left : Expression
374         right: Expression;
375 SubVectorExpression::
```

```
376         VecName: Expression
377         From: Expression
378         To: Expression;
379
380 LengthExpression::
381         SeqName: Expression;
382
383 IndexesExpression::
384         SeqName: Expression;
385
386 DistConcat::
387         SeqName: Expression;
388
389 ElementsExpression::
390         seqname: Expression;
391
392 IfExpression ::
393    if_expression : Expression
394    then_expression : Expression
395    elseif_expression_list : seq of ElseIfExpression
396    else_expression : Expression;
397
398 ElseIfExpression ::
399    elseif_expression : Expression
400    then_expression : Expression;
401
402 SetExpression = SetCompExpression |
403                                 SetDifferenceExpression |
404                                 SetEnumExpression |
405                                 SetIntersectExpression |
406                                 DistUnionExpression |
407                                 DistIntersectionExpression |
408                                 SetRangeExpression |
409                                 SubSetExpression |
410                                 SetUnionExpression |
411                                 ProperSubsetExpression |
412                                 InSetExpression |
413                                 NotInSetExpression |
414                                 PowerSetExpression;
415
416 SetCompExpression::
417                         --expression: Expression
418                         bindlist: seq of Bind
419                         predicate: [Expression];
420 SetDifferenceExpression::
421                         left: Expression
422                         right: Expression;
423 SetEnumExpression::
424                         args: seq of Expression;
```

```
SetIntersectExpression::
                        left: Expression
                        right: Expression;
SetRangeExpression::
                        head: Expression
                        tail: Expression;
SubSetExpression::
                        left: Expression
                        right: Expression;
SetUnionExpression::
                        left: Expression
                        right: Expression;
PowerSetExpression::
                        setname: Expression;
ProperSubsetExpression::
                        left: Expression
                        right: Expression;
DistUnionExpression::
                        setname: Expression;
DistIntersectionExpression::
                        setname: Expression;
InSetExpression::
                        owner: Expression
                        elm: Expression;
NotInSetExpression::
                        owner: Expression
                        elm: Expression;

MapExpression = MapCompExpression |
                                MapDomainExpression |
                                MapEnumExpression |
                                MapInverseExpression |
                                MapletExpression |
                                MapRangeExpression |
                                MapUnionExpression |
                                DistMergeExpression |
                                DomainResToExpression |
                                DomainResByExpression |
                                MapApplication |
                                RangeResToExpression |
                                RangeResByExpression ;

RecordExpression = RecordModifier |
                                    RecordConstructor;

RecordModifier::
                RecordName: Expression
                modification: seq of RecordModification;
RecordModification::
```

```
474                 fieldname: Identifier
475                 value: Expression;
476 RecordConstructor::
477                 typename: Identifier
478                 args: seq of Expression;
479
480 MapApplication::
481                 name: Expression
482                 key: Expression;
483
484 MapCompExpression::
485                 maplet: MapletExpression
486                 bindlist: seq of Bind
487                 expression: [Expression];
488 MapDomainExpression::
489                 mapname: Expression;
490 MapEnumExpression::
491                 args: seq of MapletExpression;
492 MapInverseExpression::
493                 mapname: Expression;
494 MapletExpression::
495                 left: Expression
496                 right: Expression;
497 MapRangeExpression::
498                 mapname: Expression;
499 MapUnionExpression::
500                 left: Expression
501                 right: Expression;
502 DistMergeExpression::
503                 mapname: Expression;
504 PlusPlusExpression::
505                 left: Expression
506                 right: Expression;
507 DomainResToExpression::
508                 left: Expression
509                 right: Expression;
510 DomainResByExpression::
511                 left: Expression
512                 right: Expression;
513 RangeResToExpression::
514                 left: Expression
515                 right: Expression;
516 RangeResByExpression::
517                 left: Expression
518                 right: Expression;
519 StarStarExpression::
520                 left: Expression
521                 right: Expression;
522
```

```
HeadExpression ::
        expression: Expression;
TailExpression ::
        expression: Expression;
UnaryExpression ::
        operator : UnaryOperator
        expression : Expression;

UnaryOperator =
  <PLUS> |
  <MINUS> |
  <FLOOR> |
  <ABSOLUTE> |
  <NOT>;

BinaryExpression ::
  lhs_expression : Expression
  operator : BinaryOperator
  rhs_expression : Expression;

EqualsExpression ::
         lhs_expression : Expression
     operator : BinaryObjectOperator
     rhs_expression : Expression;

BinaryObjectOperator =
        <EQUALS> ;

BinaryOperator =
  <PLUS> |
  <MINUS> |
  <MULTIPLY> |
  <DIVIDE> |
  <DIV> |
  <REM> |
  <MOD> |
  <LT> |
  <LE> |
  <GT> |
  <GE> |
  <EQ> |
  <EQEQ>|
  <NE> |
  <OR> |
  <AND> ;
---
--- STATEMENTS
---

```

```
Statement =
        ReturnStatement |
    AssignStatement |
    IfStatement |
        ElseIfStatement|
        BlockStatement |
        IsSubClassResponsibilityStatement |
        NotYetSpecifiedStatement |
        AtomicStatement |
        EmptyStatement |
        ErrorStatement |
        ForLoop|
        WhileLoop|
        ContinueStatement|
        CallStatement |
    CallObjectStatement ;

ForLoop = SeqForLoop |
                  SetForLoop |
                  IndexForLoop;

SeqForLoop::
        Reverse: bool
        VarName: seq of char
        Seq: Expression
        statement: Statement;

SetForLoop ::
        VarName: seq of char
        Set: Expression
        statement: Statement;

IndexForLoop::
        VarName: seq of char
        From: Expression
        To: Expression
        Step: Expression
        statement: Statement;


WhileLoop::
        Condition: Expression
        statement: Statement;

ErrorStatement::;

ContinueStatement::;

EmptyStatement::;
```

```
621
622 AtomicStatement::
623         statements: seq of AssignStatement;
624
625 IsSubClassResponsibilityStatement ::;
626
627 NotYetSpecifiedStatement ::
628         owner: seq of char;
629
630 ReturnStatement ::
631         expression : [Expression];
632
633 BlockStatement ::
634         statements : seq of AssignmentDefinition;
635
636 AssignStatement ::
637   state_designator : StateDesignator
638   expression : Expression;
639
640 StateDesignator =
641   StateDesignatorName |
642   FieldReference |
643   MapOrSequenceReference;
644
645 StateDesignatorName ::
646   name : Name;
647
648 FieldReference ::
649   state_designator : StateDesignator
650   identifier : Identifier;
651
652 MapOrSequenceReference ::
653   state_designator : StateDesignator
654   expression : Expression;
655
656 IfStatement ::
657   expression : Expression
658   then_statement : Statement
659   elselist: seq of ElseIfStatement
660   else_statement : [Statement];
661
662 ElseIfStatement::
663         then_statement: Statement
664         elseif_expression: Expression;
665
666 CallStatement::
667         name: Identifier
668         Args: seq of Expression;
669
```

```
CallObjectStatement ::
  object_designator : [ObjectDesignator]
  name : Name
  expression_list : seq of Expression;

ObjectDesignator =
  NameDesignator |
  NewDesignator |
  ThisDesignator |
  ObjectDesignatorExpression |
  ObjectFieldReference |
  ObjectApply;

NameDesignator::
        name: Identifier;

NewDesignator::
        New: NewExpression;

 ThisDesignator::
        This: ThisExpression;

ObjectDesignatorExpression ::
  expression : Expression;

ObjectFieldReference ::
  object_designator : ObjectDesignator
  name : Name;

ObjectApply ::
  object_designator : ObjectDesignator
  expression_list : seq of Expression;

---
--- PATTERNS
---

Pattern =
        SetPattern |
        PatternIdentifier;

PatternIdentifier ::
        identifier : Identifier;
SetPattern ::
        pattern_list: seq of Pattern;


---
--- BINDINGS
```

```
719 ---
720
721 Bind =
722         SetBind |
723         TypeBind;
724
725 TypeBind ::
726         pattern_pattern : seq of Pattern
727         type_list : seq of Type;
728
729 SetBind ::
730                 Set_name: Expression;
731
732 ---
733 --- LEXICAL ELEMENTS
734 ---
735
736 Literal =
737         NumericLiteral |
738         RealLiteral |
739         BooleanLiteral |
740         NilLiteral |
741         CharacterLiteral |
742         TextLiteral |
743         QuoteLiteral;
744
745 NumericLiteral ::
746         val : nat;
747
748 RealLiteral ::
749         val : real;
750
751 BooleanLiteral ::
752         val : bool;
753
754 NilLiteral ::;
755
756 CharacterLiteral ::
757         val : char;
758
759 TextLiteral ::
760         val : seq of char;
761
762 QuoteLiteral ::
763         val : seq of char;
764
765 Identifier ::
766         name : seq of char;
767
```

```
768 end Java
```

Listing C.1: Java AST

Appendix D

Code Generator Utilities

In this package, there are 4 classes, respectively `CGCollections.java, Quote.java, Token.java` and `Utils.java`.
`CGCollections.java` includes methods which simulate the VDM++ set, sequence and map expressions in Java.
`Quote.java` is the root class of all classes that are generated from VDM++ quote types. It simulates the quote type of VDM++ in Java.
`Token.java` includes a class named 'Token' which has the same feature as the VDM++ token type.
`Utils.java` includes some methods are not included in Java standard library but necessary for the execution of the generated Java code, such as method '`implication`' which has the same function as the vdm++ operator '=>'.
A Java documentation about Utilities package is included in CD.