Engineering College of Århus

Master of Computing Science Thesis

Connecting between VDM++ and JML

by

Carlos Manuel Gomes Vilhena

Supervisors:
Luís Soares Barbosa (Minho University)
Peter Gorm Larsen (Engineering College of Århus)

Aarhus, 2008

Abstract

This thesis aims to discuss a number of possibilities for automatic connection between *VDM++* and *JML*, in both directions The project aims at identifying the possible subsets for which this connection is possible, as well as describing in detail all the limitations encountered. It is believed that such a connection can enable *VDM++* to act as a front-end for contract-based programming, the usage of tool support from both sides and can allow different teaching perspectives with respect to formal methods. The development of a prototype proof-of-concept implementation of this bi-directional mapper based on its possible theoretical limitations is the mail goal of this work.

Acknowledgements

There are a number of persons whom I would like to thank the support and commitment. First of all, I would like to thank Professor Peter Gorm Larsen for the professional and personal support he constantly gave me and for his constant commitment in making me improve my skills. His dedication to his students is remarkable and he is a source of energy and inspiration for the ones that work with him.

I would like to thank my family for their support when I decided to come to Denmark, and for all their help during the time I spent here.

I would also like to thank Professor Luís Barbosa for his support and advice before comming to Denmark.

Finally, I would like to dedicate this thesis to Adriana Santos. I am absolutely sure that without her support, energy, patience, care and good mood, none of this would be possible. She gave me strengths when I most needed and she had patience when I least deserve it. Together we overcome difficulties and we faced challenges. For all this and much more, Thank You!

Contents

Chapter 1

Introduction

This chapter aims to present the motivation of this work, the necessary background related to the main topics of this thesis and also a number of reading guidelines to ease the reading process. Thus, in section 1.1 one can see the aim of this thesis. Furthermore, in section 1.2, the background is presented. The expected outcomes are listed in section 1.3, the necessary guidelines are presented in section 1.4 and finally the outline of the thesis is presented in section 1.5.

1.1 The Aim Of This Thesis

This thesis aims to discuss a number of possibilities for automatic conversion between *VDM++* and *JML*, in both directions, as part of a project to enable *VDM++* as a front-end for contract-based programming and the possible usage of tool support both from *VDM++* and *JML*. In particular, the project aims at identifying the notational subsets for which the envisaged automatic translation is possible, as well as describing in detail all the limitations encountered.

Thus, the development of a prototype proof-of-concept implementation of this bi-directional conversion based on its possible theoretical limitations is the mail goal of this work.

1.2 Background

In the subsequent subsections one can see the necessary background of the main topics related to this thesis. In subsection 1.2.1 one can see an introduction to *VDM++*and in subsection 1.2.2, one can find an introduction to *JML* and it main features.

1.2.1 *VDM++*

The Vienna Development Method (VDM) [4] [8] [14] [3] is one of the longest-established Formal Methods for the development of computer-based systems [16]. It is composed by two formal specification languages: *VDM-SL* and *VDM++*. *VDM++* is an extension of *VDM-SL*, which supports the modelling of object-oriented and concurrent systems [2].

For the purposes of this work, the object-oriented version (*VDM++*) will be used to create a mapping between *VDM++* and *JML*.

Along this thesis, it is assumed that the reader is familiarized with the specification language *VDM++*. However, a number of references is provided in the beginning of this subsection which can be consulted.

1.2.2 Introduction to *JML*

The *Java Modeling Language* (*JML*) [12] [5] [10] [6] [15] is a behavioural interface specification language that can be used to specify the behaviour of *JAVA* modules. Note that *JML* was not designed for specifying the behaviour of an entire program.

It combines the *Design By Contract* (*DBC*) [13] approach and the model-based specification with some elements of the refinement calculus. Because *JML* is based on the *DBC* approach, it can only use *query* methods in the assertions. The reason behind this rule is that the methods used in those assertions are requires to be side-effect free. The reserved name given by *JML* to these methods is *pure*. Thus, every method declared as *pure* is side-effect free, and then can be used in all kinds of assertions. The general idea behind *JML* is that a class and its clients celebrate a contract between them. In that contract, there are some rules that both the class and its clients must respect. The clients must guarantee a number of properties before invoking a method of a class. Those properties together are the pre-condition of the method; on other hand, the class whose method is being called must guarantee that a number of properties will hold after the call. These properties are called the post-condition of the method.

Besides this pre/post conditions, *JML* also supports invariants. Generally, invariants are predicates that hold after a sequence of operations. *JML* capture this idea, but divide the invariants in two categories: static invariants and instance invariants. These two categories are different syntactically and semantically. The static invariants are assertions that must hold of the static attributes of a class. These invariants can be identified in *JML* code with the reserved word *static*. They should not be visible from the client, and usually specifies the behaviour of the algorithm implemented. The instance invariants define the acceptable states of an object that the client can see (publics). These invariants must hold after invoking a constructor and in the beginning and end of a

method execution. Thus, the type invariant is implicitly included in the pre/post conditions and in the post-condition of the class constructor.

Some of the features mentioned in the above lead us to think that *JML* is quite similar to *VDM++* concerning to specification capabilities. The pre/post conditions have the same behaviour in both specification languages, and the instance invariant has only some slight differences. For more details about the differences between *JML* and *VDM++*, see chapter 3. However this is only the core of *JML*. In order to specify the behaviour of object-oriented models, there is a need of having more constructors, allowing other features that the core cannot handle directly. Besides the core functionalities of *JML*already described above, it is possible to use exceptional post-conditions, assignable clauses [18], model fields, ghost fields and model methods, or abstraction functions. For more information, please see [9] [19] [6] [9].

The precise semantic meaning of each one of the *JML* features listed above can be seen in section 3.2.

A key feature in the object-oriented paradigm is inheritance [17]. The possibility of reusing code, override an abstract method for a particular situation or even extend a class overriding its methods, adding more methods and use its properties is extremely useful in the object-oriented paradigm and is a part of its essence.

JML, as a specification language for the *JAVA*object-oriented programming language, also has this feature. Because *JML*is a behavioural interface specification language, it is important to assure the inheritance of the specification cases presented in a class. Thus, each subclass of a superclass must respect a contract celebrated between them. The superclass invariants must be valid also in the subclasses, and for each overriding method implemented in a subclass, it has to meet the overridden method specifications specified in the superclass. The invariants presented in the superclass are applied to each subclass, so the invariants are preserved in the inheritance tree. To let the overriding method specification to meet the overridden method specification, there is a need of using a reserved keyword in the JML named *also*. Using the reserved keyword *also* in the subclass to combine the two specification cases of the overridden and the overriding methods, they are conjoined and form a unique specification. However, effectively this is not a conjunction. There is a difference between the treatment given to the pre-condition and the post-condition. The pre-condition of the overriding method will be a disjunction of the pre-conditions from both specification cases. The post-condition will be a conjunction of implications. In each implication, one pre-condition implies the correspondent post-condition.

1.3 Expected Outcomes

After the conclusion of this work, it is expected to include in the Overture Tool the capability of connecting between the VDM++ and the JML, in a bidirectional way.

Besides the connection itself, it is also expected to have a solid theoretical background sustaining the referred connection and the possible limitations encountered along this work. A detailed exploration of the possible subsets in which that connection is possible should also be performed in order to understand what constructs can or cannot be used from both specification languages.

1.4 Reading guidelines

In order to ease the reading process, one can see the reading guidelines presented below.

Whenever the symbol $^{\tau}A$ appears, it means that the word A is explained in the terminology chapter.

Furthermore, there are three types of code along this thesis: *VDM++* code; *JML* code and finally AST code representing Abstract Syntax Tree definitions written in *VDM-SL*. Each of these three different possibilities has different layouts as it can be seen below:

- *VDM++*

```
public op : nat * seq of char ==> ()
op(n,s) == is not yet specified;
```

- *JML*

```
/*@
  @ requires x > 1;
  @*/
```

- AST

```
public PairType ::
  fst : Type
  snd : Type;
```

Moreover, the tables and figures are referred in the list of tables and the list of figures, respectively., which are in the end of document.

1.5 Outline Of Thesis

In chapter 1, the introduction to this project is presented. The conceptual analysis is described in chapter 2. Furthermore, the correlation between *VDM++* and *JML* can be seen in chapter 3. The preliminary design of the proposed connection is presented in chapter 4. Chapter 5 contains the description of the specification of the bidirectional mapper between *VDM++* and *JML*; chapter 6 presents the case studies selected for this thesis and finally, chapter 7 contains the conclusion of this thesis.

Concerning the appendixes, appendix A contains the abstract syntax definition of *JML*. Furthermore, appendix B contains the complete specification of the mapper, in *VDM++*. Finally, in appendix C it is possible to see the specification and results of the case studies.

Chapter 2

Conceptual Analysis

2.1 Usage and Applicability of *JML*

The *JML* descriptions can simply be inserted in a large variety of files and are written in special *JAVA* comments, similar to traditional Javadoc [10]. The syntax used by the *JML* is very similar to the *JAVA* syntax. Besides some reserved words used to express the semantic meaning of the *JML* features, the leftovers of the syntax is the same as the *JAVA* syntax. This allows a quicker introduction of *JAVA* programmers to this specification language.

Taking into consideration all the features presented in the *JML* and explained so far, it becomes necessary to present the reasons why one should use this behavioural interface specification language. After making a thorough analysis of the *JML*, one can say that there are two major good reasons to use it [6]:

1. Precise and unambiguous specification of the behaviour of *JAVA* modules and documentation of *JAVA* code;

2. Tool support available [11].

Each of the two reasons presented above will now be analysed carefully.

Analysing the first reason, it states that *JML* is a precise and unambiguous specification language for the behaviour of *JAVA* modules and documentation of *JAVA* code.

First of all, there are two different roles that the *JML* can take here: it can be used as a behavioural specification language for *JAVA* modules; it can also take a role as documentation annotations for *JAVA* code. Using the *JML* as a specification language, one can see that it establishes *contracts* between classes/interfaces, which act as controllers between the interactions of classes/inter-

faces. The invalid use of objects, classes or methods is forbidden by clauses, and these can be checked at runtime.

Using *JML* as formal documentation could also an advantage, considering the following sentence. One can separate the code from the specification, and then we can see how the contract is established clearly, opposing to the ambiguous natural language of common *JAVA* documentation. There are users (e.g. those not familiarized with software engineering) that do not care about implementation details, but they care about the specification issues evolved. In this way, they can have a clear overview of the behaviour of *JAVA* classes/interfaces.

It is also possible to analyse in a formal way certain properties or a design, perform formal verification or reasoning about the correctness of the code in a more clear fashion. Secondly, the variety of tool support available is also an strong area for the *JML*.

JML has a variety of tool supporting features such as parsing and type checking, static analysis, formal verification, recording of dynamically obtained invariants, runtime assertion checking and unit testing [11].

After observing these two reasons presented in the text above, it is possible to conclude that *JML* is a strong specification language, with similar semantic features of the *VDM++* specification language, which lead us to a first expectation that a sensible connection between *JML* and *VDM++* should be possible to obtain.

2.2 Object-Oriented Key Features

2.2.1 Inheritance

One of the most important concepts in the *Object-Oriented paradigm* is inheritance. Both Java (with and without JML assertions) and VDM++ are Object-Oriented languages playing different roles: while VDM++ is used for modelling purposes [4], Java is used to implementation purposes. Besides this slight difference, there are also some differences concerning to the usage of the concepts of the Object-Oriented paradigm. One of those differences is inheritance.This concept is used differently in Java with JML assertions and VDM++, and the objective of this section is to explain those differences.

Java is strict concerning inheritance in the sense that it does not allow multiple class inheritance. However *JAVA* does allow multiple interface inheritance. The reason behind this restriction of multiple class Inheritance is due to the basic principles of Java design. It is believed that multiple class inheritance causes more problems than it actually solves and thus conflicts with one of the Java rules which states that Java should be simple and familiar. On the other side, VDM++ allows multiple class inheritance. This is a very serious challenge, if one is considering connecting VDM++ with JML. If one has a model in VDM++

that takes advantage of multiple Inheritance, and wants to move it to JML, it is necessary to have an answer to this problem.

The adopted solution will be promote all classes but one to interfaces, when moving from *VDM++* to *JML*. This way, if such conversion is possible, the multiple inheritance problem is solved. On the other hand, if it is not possible to convert the necessary *VDM++* classes to *JAVA* interfaces, the user must change the specification in order to continue.

Concerning to specification inheritance, both JML and VDM++ have similar behaviour. The specification inheritance in JML guarantees that all subtypes are behavioural subtypes. Thus, all subtypes inherit the specifications from its supertype, including private specifications and fields, although they are not visible. For more details about visibility, see section 2.2.2.

All the method assertions are also inherited under the same principles of visibility mentioned above and the supertype invariant must also hold in the subtype.

Each assertion presented in a JML supertype must be in conformance with the assertions present in the subtype, building stronger conditions for the subtypes.

In VDM++, the inheritance comprehends instance variables, invariants and their restrictions on the allowed modifications of the state (visibility), operations and functions (including assertions, e.g. Pre-conditions), value and type definitions and at last synchronization definitions.

2.2.2 Visibility

There is a need of comparing the visibility between *VDM++* and *JML*. However, since *JML* is a specification language for *JAVA*, there is also a need of study the visibility rules of *JAVA* and understand how they are related, if so.

In *VDM++* the attributes can be divided in two types: class attributes or instance attributes. Those attributes can be functions, operations, instance variables or constants. Moreover, the type construct is always a static attribute; on the other hand, the thread and the synchronization constructs are always instance attributes.

The default attribute type for *VDM++* constructs is instance, and if ones want to specify a static attribute must use the keyword *static*.

Concerning the accessibility, it may be explicitly defined using three different keywords [7]: public, private and protected. Those three constructs are explained below.

Public: Any class may use such members, with no restrictions;

Private: No other class but the one where the definition is written may use those members;

Protected: Only subclasses of the class where the definition is written may use those members.

Finally, the default value for accessibility in *VDM++* in case there is no explicit definition is private.

Before starting the explanation about visibility in *JML*, it is important to see *JAVA* rules about this issue. In *JAVA*, there is also a separation of attributes in instance attributes or class attributes. The semantic meaning of this two types is the same as in *VDM++*.

Concerning the accessibility, besides the three constructs defined above which have the same semantic meaning in *JAVA*, there is also another access modifier called *default*. This modifier, also known as *package private*, states that a given attribute is accessible only inside the package where it is defined. This is the default value when no other is used, thus there is no keyword associated to it.

On top of *JAVA*, it is possible to append *JML* assertions, which imposes extra rules in addition to the *JAVA* rules about visibility.

Any *JML* assertion such as an invariant or a method specification cannot refer to names that have a more restrict visibility that the current context where the name is being invoked. Thus, a reference to a specific name is valid if the visibility of the context of the assertion which contains the reference to the name is at least permissive as the declaration of the referred name itself.

Finally, each *JML* accessibility rule must hold, and after that, also the *JAVA* accessibility rules must hold in order to state that the visibility in *JML* is being respected.

2.3 A Logic Behind the Connection

Establish a bidirectional connection between the *VDM++* and the *JML* needs a clear and concise explanation about its purposes. Indeed, it is believe that this connection can possible bring advantages to software developers wiling to use it. Moreover, all the items presented bellow form a sustainable basis for the construction of such a connection and feed the motivation to do so.

2.3.1 Teaching perspective

From an educational point of view, this connection can be seen as a bridge between *VDM++* and *JML* in both directions. For example, to teach *VDM++* to students or software developers with a Java background, one may start with using *JML* assertions inside Java programs, and thus move such specifications to *VDM++*. On the other hand, it is possible to use *VDM++* as a front-end for contract-based programming. For Java students with familiarity with *VDM++* this

connection may be of use to move *VDM++* specifications into *JML* annotations as a starting point for Java development.

2.3.2 Tool support

Sharing the tool support available from *VDM++* and *JML* communities will certainly extend the range of supply tools and help in a positive way those who want to use this features. If there is a tool support available at one side, which is not available at the other, one could move the specifications from one side to another, take advantage of the tool support, and then return with the specifications to the starting point. This could also be advantageous if the tool support is better at one side than another.

2.3.3 Java code generation

The automatic generation of *JAVA* classes and interfaces from formal *VDM++* specifications could also benefit from the proposed connection. Since the JML is a behavioural interface specification language for describing in a non ambiguous way the behaviour of *JAVA* modules, it can be combined with the *JAVA* code generator to generate classes/interfaces that should control the behaviour of the *JAVA* modules generated.

Thus, because *JML* assertions appear on the form of Java comments, it is possible to compile them, transforming *JML* assertions into executable *JAVA bytecode*. Then, the execution of the system is regulated by those assertions. Furthermore, *JML* assertions can also be compiled using a regular *JAVA* compiler. In this way, *JML* assertions are nothing more than *JAVA* comments, and have no effects over the execution of the system.

2.3.4 Front-end for contract-based programming

Establishing this connection enables the use of *VDM++* as a front-end for contract-based programming. Hence, it is possible to connect formal specifications of object-oriented systems to behavioural specifications of object-oriented interfaces.

2.4 Connecting Between *VDM++* and *JML*: Inheritance Approach

Considering that *JML* offers a large variety of different specification approaches [5], it is necessary to analyse each one carefully and consider which one should be used in this specific situation of connecting *VDM++* and *JML*. Before going to this subject in detail, it is important to notice that there are some key features

presented in *JAVA* that must be well known in order to better understand the decisions that are about to be made. For those who are not familiarized whit the concepts presented below, it is recommended to refer to appendix C.4.1. Concerning *JAVA* constructs, it is important to be familiarized with *Abstract Classes*, *Classes* and *Interfaces* [17]. The discussion presented below focus on those three *JAVA* constructs. Besides those constructs, it is also important to understand some of the key principles of the Object-Oriented paradigm such as inheritance and visibility. Those principles are also explained in the terminology appendix.

As one can see on the referred constructs, these three different concepts in the *JAVA* programming language play different roles. For each of them, it is possible to append the *JML* assertions, because they are simply *JAVA* comments, from a *JAVA* perspective. Therefore, a description of each possible alternative can be found below, with their pros and cons, in connection to the ability to connect between *VDM++* and *JML*. Such a description will be based on specific items: variables, constructors, methods and inheritance. Each of those items will be explained for each possible approach, in order to make a well-funded decision.

2.4.1 Using *JAVA* Abstract Classes to specify *JML* assertions

In this section, the possibility of writing *JML* assertions inside *JAVA* abstract classes will be studied. The pros and cons will be carefully consider in order to decide in a further stage what is the best approach for this concrete situation.

Concerning variables, the abstract classes does not have any particular limitation. One can create variables and thus it is possible to append *JML* assertions to them. A possible future implementation would have the variables already specified, thus it would not be necessary do re-define them. In an abstract class, although it is not possible to instantiate it, it is possible to have constructors. Furthermore, if a *VDM++* specification contains a constructor, it would be possible to write it in *JAVA* notation inside the referred class, and also append whatever assertions it has in *VDM++* as *JML* assertions over it. Thus, the constructors offer no limitations to this approach. Concerning the usage of methods in an abstract class, there are a few limitations, that do not affect the intended usage of those classes. An abstract class can have a number of methods, and some of those methods (at most all but one) can also have bodies. Thus, each header of the methods presented in a *VDM++* specification can be translated to *JAVA* headers and all the *VDM++* assertions connected to each method can also be translated into *JML* assertions.

Reasoning, the three items explained above causes no limitation to a possible usage of abstract classes, but there is another item to discuss: inheritance. As it can be seen below, although *JAVA* allows multiple inheritance of interfaces,

it does not allow multiple inheritance of classes. This concrete limitation affects this possible approach of using abstract classes in this connection. The *JML* assertions among with some other specific code would be written in an abstract class, and thus the implementation should extend that specific class. Moreover, if the implementation needs to extend another class, it is not possible because it is already extending one: the class containing the specification. This is a serious handicap, because it disables the possibility of having class inheritance in the *VDM++* model, which is not one of the expected outcomes of this connection.

2.4.2 Using *JAVA* Classes to specify *JML* assertions

The use of *JAVA* classes to specify *JML* assertions is very similar to the one presented above. The variables are dealt in the same way, *i.e.*, they would not cause any difficulty using this approach. The same happens to the constructors. They can be written in the *JAVA* class with their assertions with no loss. Concerning the inheritance issue, the same problem happens here. It is only possible to have single class inheritance in *JAVA*, thus the limitation presented above also applies to this approach. There is no intention in prohibit the usage of single inheritance in the *VDM++* model, thus this is a serious obstacle to this approach. Other obstacle of this approach lies with the methods. What differs a class from an abstract class is the obligation of the class to have all the methods with bodies, *i.e.*, the implementation should be in the same place as the specification. This would not be a problem if the aim of this project was to translate not only specifications but also implementations, as a *JAVA* code generator. Considering that the aim of this thesis is connecting no more than *VDM++* and *JML* assertions, this approach does not make sense.

2.4.3 Using *JAVA* Interfaces to specify *JML* assertions

If the *JML* assertions will be written in an Interface, there are some limitations, as it can be seen by the interface definition in appendix C.4.1. In interfaces, it is not possible to create variables, except static final variables. This is clearly a handicap of the Interface specification, but *JML* can handle this situation. Using model fields, it is possible to declare model variables that can represent the real variables in a future implementation. Thus, when the concrete *JAVA* implementation will be built, each concrete variable must have a *JML* assertion stating which model variable from the specification is being implemented. This way, the concrete variable will be associated with the specification variable, and thus the assertions connected to the specification variable (e.g. invariants) will also apply to the implementation variable. Moreover, it is not possible to have constructors in an interface. If a *VDM++* specification has a constructor with a pre-condition stating that the invariant must be preserved, it can not be written in an interface.

Again, the *JML* can handle with this situation. The possibility of declaring public invariants, override the referred problem. Although the constructor cannot be declare, and thus the pre-condition can not be there, the simple conversion of the constructors pre-condition to the public invariant solves the problem. Thus, when the implementation is completed, the constructor will be affected by the public invariant, and each object of that type must respect the invariant. Finally, for each method presented in *VDM++* specification, only the header can be written in the interface. If the method has assertions associated, they can also be written as *JML* assertions on top of the methods headers with no loss. Furthermore, when one complete the implementation, each method of the concrete implementation will have to respect the assertion present in the interface specification.
Concerning the specification inheritance, each method that can possible override one method from the implementation will also have to respect the assertions specified in the interface specification.

This approach will lead to a decision, in order to connect each *VDM++* basic and compound types. There are two possibilities to use as models fields: *JML* pure types [6] and *JAVA* types. Only one of this two alternatives should be used to be mapped to the already existing *VDM++* types [2], in order to have concistency in the connection between *VDM++* and *JML*. This discussion can be found below, in section 2.4.4. Moreover, there is no limitation concerning the inheritance. Because *JAVA* allows multiple inheritance of interfaces, a possible implementation or code generation to *JAVA* of *VDM++* specification will have to implement the interface created, in order to continue respecting the assertions, and can also implement other interfaces, if the user wants, with no loss at all. A resume of all the above possibilities and its pros and cons can be found in table 2.1.

-	Variables	Constructors	Methods	Inheritance
Class	√	×	×	×
Abstract Class	√	√	√	×
Interface	√	×	√	√

√ - No limitations detected
× - Limitations detected

Table 2.1: Comparison between possible *JML* implementations using inheritance.

Reasoning, the approach that produces less consequences to the implementation of *JML* is the one that uses interfaces to specify *JML* assertions. Therefore this approach will be taken into account in further discussion.

2.4.4 *JML* pure types vs *JAVA* types

In *VDM++*, it is possible to use in each predicate (*e.g.* pre-condition) any pre-defined construct that gives as a result a boolean expression. Thus, it should be possible to maintain this feature when moving specifications from *VDM++* to *JML*, in order to maintain consistency between the semantic value in both sides. *JML* is more restrictive than *VDM++* with respect to the usage of methods in assertions, such as invariants, pre-conditions and post-conditions. It is only possible to use pure methods inside assertions, in order to avoid possible side effects that such methods could create. As it can be seen above, there are two possibilities to use as model fields: *JML* pure types and *JAVA* types. Because *JAVA* types does not give any guarantee of its purity, using them it is not a good solution for this situation. The methods associated to a specific type can provoke side effects, thus they cannot be used in *JML* assertions.

Reasoning, the purity of *JML* types allows the maintenance of the possibility of using methods in assertions, preserving the semantic of such *VDM++* predicates. Yet, there is one limitation using these types. *JML* pure types are model types, meant to be used for specification purposes only. Thus, these types cannot be used for implementation purposes. This means that if a specification uses *JML* pures types, an implementation of it cannot use those types. Instead, *JAVA* types must be used. Consequently, there must be a connection between the specification types and the implementation types, otherwise the assertions present in the specification will not be applied to the implementation. Although *JML* as a specific construct named *represents* responsible for connecting types present in the specification to types present in the implementation, this can only be used when those types are the same. If the types are different, when one try to use the referred clause, the *JML* type checker will generate a type error. The solution for this is to map each *JAVA* type used in the implementation to a correspondent *JML* pure type present in the specification through a model method written inside the implementation. This way, a specific *JAVA* type will be connected to a *JML* pure type and the variable will receive its specifications. This approach will grant the possibility of having a number of implementations for one specification, each one with a model method mapping the value of a concrete variable to the correspondent abstract variable.

The syntactic mapping between *VDM++* types and *JML* types can be found in chapter 3.

2.5 Connecting Between *VDM++* and *JML*: Refinement Approach

An alternative approach from the one presented in section 2.4 takes advantage of *JML* definition of refinement [6]. It is possible in *JML* to define specifications

both inside *JAVA* modules or in different files, separating the specification from the implementation. This separation takes place due to a number of reasons: the implementation is not available; the specification is meant to be design first; the *JAVA* implementation is in a binary format, among other possibilities explained in [6]. Since this project aims to connect *VDM++* and *JML*, the specification will be generated before having an implementation. Thus, the alternative solution explained below will be based on this assumption. The possibility of separating the specification from the implementation is provided by *JML* using the definition of refinement. Each file containing *JML* specifications is connected to the implementations using a refinement construct, establishing the relation between them. There are also rules to be respected using this alternative described in [6]. With respect to this connection, only one of those rules will be explained due to the relevance of it for further decisions. All the other rules will not be explained and will only be considered in the implementation of the proposed connection. The rule referred above states the following:

> Non-model methods or constructors can only have bodies in the *JAVA* file, and never in specification files, otherwise an error will occur.

This means that each method or constructor must only have a body in the implementation. However, each method can have a chain of specifications, all connected by a refinement clause, considering the number of specification files existing.

Just like in the previous alternative, a decision must be made concerning the specification approach. Since *JML* assertions can be written inside *JAVA* classes, abstract classes and interfaces, one should be used in this connection. Although there is a discussion in the previous section about the pros and cons of each alternative, that discussion is not relevant for this alternative approach because of two issues: inheritance and the rule highlighted above. In this approach, the inheritance is not used as a means of transferring specifications, thus classes and abstract classes can be used without inheritance limitations. One consequence of the rule mentioned above consists in allowing a specification file, which must be a class, abstract class or interface, to have an incomplete implementation. As a result, one can have a *JML* specification without the body of the methods written in a *JAVA* class, without having an error reported.

Regarding what types should be used in this approach, *JML* pure types or *JAVA* types, the choice is the same as in the previous subsection, *i.e.*, *JML* pure types, and the reasons behind this choice can be found in 2.4.4.

Reasoning, if one wants to use a class to write *JML* assertions will not yield any problems. The methods could be empty, and all the other functionality is present. If one wants to use abstract classes, there are also no limitations. Finally, if one wants to use interfaces, there will be limitations. It will not be possible to write the constructors headers, because interfaces can not yield constructors,

nor its assertions. Considering this limitation, and considering also that the other two solutions does not present limitations, using interfaces will not be a satisfactory solution for this approach. The full resume of the pros and cons each *JAVA* construct can be found below:

-	Variables	Constructors	Methods	Inheritance
Class	√	√	√	√
Abstract Class	√	√	√	√
Interface	√	×	√	√

√ - No limitations detected
× - Limitations detected

Table 2.2: Comparison between possible *JML* implementations using refinement.

From the table 2.2, it can be seen that both classes and abstract classes can be used without any semantic complications in this connection. However, only one should be used. In order to maintain the semantic value of *VDM++* classes, *JAVA* classes will be used to hold *JML* assertions and thus to map *VDM++* specifications.

2.6 Connecting between *JML* and *VDM++*

Concerning the connection from *JML* to *VDM++*, there are no semantic limitations, excluding construct differences. Thus, there are no inheritance problems in this connection, and because *VDM++* has only classes, each *JML* class, abstract class or interface will be mapped into a *VDM++* class. This will make the specification of this mapper a straight forward process, where only the construct limitations will be taken into account.

Chapter 3

Correlation Between *VDM++* and *JML*

All through this chapter, the correlation between *JML* and *VDM++* will be presented in a semantic perspective. Since the intention of the proposed connection is to map *VDM++* and *JML* specifications in both directions, a semantic comparison between those two specification languages must me carried through. Such comparison should be divided in logical items, in order to make sure that all the relevant constructs and features are correctly related in order to be possible a mapping between *VDM++* and *JML*. For this purpose, two main items were selected, and are briefly explained below. Each of those items will be analysed through this chapter, in order to show the correlation between those two specification languages

- **Types** Since both *JML* and *VDM++* have pre-defined types, the correlation between those types must be analysed, in order to correctly proceed to the mapping between them;
- **Constructs** Each predefined type has its own constructs associated. Thus, for each type relation proposed above, its constructs should also be related in order to map them.

3.1 *VDM++* types vs *JML* pure types

In this section, the correlation between *VDM++* types and *JML* pure types will be analysed. Furthermore, a mapping will be established between those types, taking into account the semantic meaning of each type. Moreover, the possible limitations of such mapping between types will be analysed in this section.

There are two kinds of types in *VDM++*: basic and compound types. This dissociation between basic and compound types will be used in this section in order to help along the read process.

3.1.1 Basic Types

The *VDM++* basic types will all have correspondence to *JML* types. However, in order to have equivalence to the *VDM++ nat* type, a new type was created in *JML*, named JMLNatType. This new type will receive as input to the constructor the natural number 0 or 1, depending if the mapping is being performed from a *nat* or a *nat1* type. The correlation between the types is as follows, in table 3.1:

VDM++ type	JML type
bool	boolean
nat	JMLNatType(0)
nat1	JMLNatType(1)
int	JMLInteger
real	JMLFloat
char	JMLChar
token	JMLType
Quote	JMLEnumeration

Table 3.1: Comparison between types.

As it can be seen from the table above, the *VDM++* simple types are completely mapped into *JML* types.

3.1.2 Compound Types

Concerning the compound types of *VDM++*, the correlation with the *JML* types can be see from table 3.2. Again, there are new types created an the *JML* side to deal with a number of translations. Those types are *JMLClassType* and *JMLTuple*.

The JMLClassType was created to be mapped to the *VDM++* composite type. On the other hand, the JMLTuple was created to be mapped to the *VDM++* product type.

Record types in *VDM++* are converted into classes at the *VDM++* level, due to the fact that they have a number of fields with possible different types. Thus, whenever a record is created in *VDM++*, a *JML* class is created where each field from the *VDM++* specification is converted into a instance variable. Furthermore, if the product type has an invariant, it can be translated into the instance invariant of the class.

VDM++ type	JML type
Composite	JmlClassType
Product	JMLTuple
Set	JMLValueSet
Seq	JMLValueSeq
Map	JMLValueToValueMap

Table 3.2: Comparison between compound types.

3.2 Language Semantics

3.2.1 *VDM++* Pre-Condition vs *JML* Requires Clause

Both the pre-conditions and the requires clause are used to specify what should hold before the invoked method or function starts its execution. They are therefore composed by a truth-value expression that can only refer to the inputs of the given method or function and to global variables in general in case of operations. If the expression evaluates to true, then it is possible to proceed with the execution, otherwise there will be a violation of the pre-condition and therefore the execution of the method is aborted. In fact, at the *VDM++* side it is possible to proceed with the execution of a function or method in which the pre-condition has evaluated to false, deactivating that option. However, disabling that option leads to a lack of guarantee about the results of the execution. It is possible that, for a number of inputs, a given method or function will abort its execution, because there are no conditions to validate that input.

The behaviour of both referred constructs is similar. However, there are a number of items to compare, in order to build a proper mapping between those two constructs. First, the structure of both constructs will be presented, and their differences analysed.

In *VDM++*, each method and function can only have one pre-condition. That pre-condition is composed by the keyword *pre* followed by a *VDM++* expression whose type must be boolean. Below, it is possible to see the syntactic form or the referred operator:

pre Expression;

The absence of the pre-condition leads to a default value of *'true'*, meaning that each input for a given method or function is accepted.

Concerning the *JML* requires clause, as it can be seen above, its behaviour is similar to the *VDM++* pre-condition. Here is the syntactic form of this construct:

requires Expression;

One exception is that for each method, it is possible to specify not only one but a number of requires clauses. However, all the requires clauses are conjuncts in a single requires clause, so semantically it is the same as in *VDM*++.

requires P;
requires Q;

It is the same as:

requires P && Q;

When there is no requires clause defined, the default value is set to *'true'* for a heavyweight specification and *'not_specified'* for a lightweight specification. Besides that, it is also possible to define the expression with another *JML* construct called
'same'. However, the usage of this construct requires that the corresponding method is overriding another, in order to use the requires clause already specified in the overridden method. Otherwise, it is not possible to use such construct. Finally, there are more constructs to represent a pre-condition in *JML*. Instead of writing the construct *'requires'*, it is possible to use three other constructs: *'requires_redundantly'*, *'pre'* and *'pre_redundantly'*. The semantic meaning of the construct *'pre'* is the same as the *requires clause*, however the *'redundantly'* constructs are slightly different. Semantically, those constructs are meant to rewrite the existing requires clauses in a way that they would become readable from a user perspective. Moreover, if there is a requires clause P and a redundant requires clause PR, then the redundant clause follows from the requires clause, *i.e.*, P implies PR:

requires P;
requires_redundantly PR;

Then, P $\Longrightarrow$ PR, *i.e.*, whenever P is true, PR must be also true. If there are a number of redundant clauses, they are conjoint in one single redundant clause like the requires clause presented above.

3.2.2 Map *VDM*++ Post-Condition to *JML* Ensures Clause

As in the previous subsection, the *VDM*++ post-condition and the *JML* ensures clause are similar. They are used to specify what should hold after the execution of the corresponding method or function. For that purpose, they use an truth-value expression that can access the input of the method or function, the current value of the global variables, the result identifier and the old value of global variables (if it is a method and not a function). If a given method or function finish its execution without throwing an exception (subsection 3.2.3), then the

post-condition must hold. If the post-condition holds, the method or function ends correctly, otherwise the post-condition was violated and thus the execution is aborted.

Below, a comparison is provided between the two referred constructs concerning a number of items, in order to guarantee that the mapping between them can be performed. As it can be seen above, the two constructs are semantically very similar. This is also the case syntactically. The syntactic form of the *VDM++* construct *post* is:

post Expression;

The expression can be any *VDM++* expression as long as it returns a truth value. In case the construct is missing, the default value *'true'* is used. Finally, only one post-condition is allowed for each method or function.

The syntactic form of the *JML* construct *ensures* is:

ensures Expression;

As in the *VDM++* post-condition, the expression can be any *JML* expression that returns a truth value. Concerning the default value when the construct is missing, it depends if one is defining a lightweight or heavyweight specification. If one is specifying a lightweight specification, the default value is *'not_specified'*; otherwise, the default value is set to *'true'*. In oppose to *VDM++*, each method in *JML* can have a number of post-conditions. However, this is the same as having only one post-condition where all the ensures clauses are conjuncts in a single ensures clause. In essence, if one has the following two ensures clauses:

ensures P;
ensures Q;

They are conjoint in order to form one ensures clause:

ensures P && Q;

As in the pre-condition, there are more constructs in *JML* to represent an ensures clause: *'post'*, *'post_redundantly'* and *'ensures_redundantly'*. The meaning of those constructs is analog to the one defined in subsection 3.2.1.

3.2.3 *VDM++* Exception Clause vs *JML* Exceptional Post-Condition

The two constructors presented in this section, the *errs* and the *signals* clauses, are the responsible for dealing with exceptional behaviour in VDM++ and JML, respectively. In order to connect them appropriately, their semantic and behaviour must be carefully analysed.

In VDM++, the errs clause can be used to describe how an operation should deal with error situations. It provides the user with a clean way of separating the normal cases from the exceptional cases. Furthermore, this clause show exactly under which condition an error can occur and what are the consequences for the result of calling the operation, but does not give information about how exceptions are to be signalled.
This *errs* constructor should be followed by a list of conditions, each one describing a specific error situation. A condition is composed by a number of elements explained below:

- An identifier, illustrated by *CONDi* (*i* in {1,...,n}), which describes the kind of error that can be raised;
- An error pre-condition, represented by *Ci* (*i* in {1,...,n}), that describes under which condition the respective error should occur;
- An error post-condition, illustrated by *Ri* (*i* in {1,...,n}), which represents the consequences for the result of calling the correspondent operation.

Syntactically, the pre-condition, the post-condition and the referred clause are different, and are defined using different constructs, however semantically they are in some specific way conjoined into one pre-condition and one post-condition as it can be seen below.
Considering an operation with pre-condition *Pre*, a post-condition *Post* and an *errs* clause, the real pre-condition of the operation will be:

$$Pre \quad \vee \quad C1 \quad \vee \quad \ldots \quad \vee \quad Cn$$

And the real post-condition of the operation will be:

$$Post \quad \wedge \quad (C1 \Rightarrow R1) \quad \wedge \quad \ldots \quad \wedge \quad (C1 \Rightarrow R1)$$

If the *errs* constructor is not used, then the pre- and post-conditions will be themselves, and the conjunctions mentioned above will not take place.
In *JML*, the *signals* clause is responsible for specifying the exceptional post-condition, *i.e.*, the property which is guaranteed to hold at the end of a method invocation when this method terminates abruptly by throwing an exception [6]. Note that this clause specifies under which conditions a certain exception may possible be thrown but when a certain exception must me thrown. For this specific situation, one should use other construct provided by *JML*, named *signals_only*.
As one can see from above, the *signals* clause is composed by the construct name followed by an exception *e* of type *E* and finally a predicate *P*.
The above syntactic form of the *signals* clause is also equivalent to the following syntactic form, which will be useful for further explanation below:

signals (java.lang.Exception e) ((e *instanceof* E) $\Rightarrow$ R)

The semantics of this constructor is presented below.
When a given method with a *signals* clause associated terminates abnormally, throwing an exception of type *E*, then in the final state of the exception object *E* the predicate *P* must hold [6].
Moreover, there are some other restrictions about the evolved types and variables. The Exception *E* must be a subclass of the Java class Exception (java.lang.Exception) and the variable e is bound in the predicate *P*.
In case the Exception *E* is an exception that does not inherit from java class RuntimeException (*i.e.* checked exception), then it must be one of the exceptions listed in the throws clause of the method, or a superclass or subclass of such declared exception.
Beyond the semantic presented above, there are two more possible situations evolving the *signals* clause which semantics and behaviour must be explained: the absence of the clause and the presence of more than one clause referring to the same method.
Beyond the semantic present above, there are two more possible situations evolving the *signals* clause which semantics and behaviour must be explained: the absence of the clause and the presence of more than one clause referring to the same method.
The absence of the *signals* clause leads to a default behaviour. If the specification is a lightweight specification, it is used the default value *not_specified*. This means that there is no treatment if the method ends abnormally. Otherwise, if the specification is a heavyweight specification, the default value for the *signals* clause is:

signals (Exception) *true*

Finally, if there are several *signal* clauses related to the same method, *i.e.*:

signals (E1 e) P1;
...
signals (En e) Pn;

This means that those clauses will be merged into one clause, and its predicate *P* will be a conjunction of all predicates of the different *signal* clauses:

signals (Exception e) ((e *instanceof* E1) $\Rightarrow$ P1) &&
... &&
((e *instanceof* En) $\Rightarrow$ Pn)

3.2.4 Map *VDM++* Invariant to *JML* Invariant

Invariants are properties in the form of expressions that must be preserved in order to guarantee the consistency of the model. They represent universal properties over the model that restricts its domain in order to avoid malfunction or inconsistency. Both *VDM++* and *JML* allow the usage of invariants, however with a number of differences that will be explored along this subsection.

In *VDM++*, invariants can be used either combined with type definitions or associated to instance variables. Either way, they are meant to limit the possible values allowed by the type or variables in which the invariant refers to. However, the syntactic form of the invariants related to types is different of the one used for instance variables. The syntactic form of the *VDM++* invariants related to types can be found below:

$$inv \quad pattern \quad == \quad expression;$$

The *pattern* is used to match to a value of a particular type. It can be an identifier or an explicit value and it is used to create a scope to the invariant. The *expression* can be any *VDM++* expression build over the *pattern* that returns a truth value. Each *VDM++* type can have one invariant, and to use the invariant of a particular type *T* elsewhere, it can be used through the expression *inv_T*. If there is no invariant associated to a type, the default value is set to *true*, which means that the domain of the correspondent type is not limited, allowing all it possible values. Below, one can see the syntactic form of *VDM++* invariants related to instance variables:

$$inv \quad expression;$$

Inside the instance variables block, one can define a number of invariants that can define properties over the instance variables declared inside the referred block. Unlike the invariant expression within the type information, these invariants do not need to define a pattern in order to be used in the invariant expression; it uses directly instance variables inside the expression to formulate the desire properties. Furthermore, the overall invariant of a *VDM++* class is a conjunction between the invariants of the superclass and the invariants of the class. If a given class has a number of invariants and one wants to access the complete invariant of the class, it is possible to use a built-in operation called *inv_classname*. In case of absence of invariants, the instance variables are not limited and thus it is possible to use all the values its types allow to be used. Finally, *VDM++* does not use access modifiers connected to invariants. Thus, each invariant can specify properties from public, private and protected instance variables all together.

In *JML*, invariants are properties that have to hold in all visible states [6] of a class. However, *JML* distinguish between two different kinds of invariants: static

and instance invariants. Static invariants may only refer to static fields of an object and instance invariants may refer to both static and non-static fields [6]. Besides the static construct, each invariant have an access modifier associated. Each invariant can be public, private, protected or have the package visibility if they do not have any of the others access modifiers. As a consequence, each invariant can only use fields or pure methods with at least the same visibility as the invariant. Furthermore, all the invariants must be preserved regardless their access modifiers. This means that, for example, a public method must preserve a private invariant. Below, it is possible to see the syntactic form of a *JML* invariant:

modifiers *invariant_keyword* expression;

The *modifiers* are an optional field and it is possible to have a number of them. There is a number of possibilities, however for the purposes of this work it will be considered the following: public, protected, private, abstract, static and final. For further details about all the possible options and access rules see [6]. Concerning the invariant keyword, like the pre and post-conditions, it is possible to have both *invariant* and *invariant_redundantly*. The expression can be any *JML* expression which returns a truth value. In order to proceed with further explanations, three different concepts, that will be used in this subsection, will be explained: assuming, establishing and preserving an invariant. The definitions as in [6] are presented below.

Assuming an invariant If the invariant must hold in the pre-state of a method or constructor, then that method or constructor assumes the invariant;

Establishing an invariant If the invariant must hold in the post-state of a method or constructor, then that method or constructor establishes the invariant;

Preserving an invariant If the invariant is both assumed and established by a method or constructor, then that method or constructor preserves the invariant.

JML invariants should also hold in an exceptional situation, *i.e.*, when a method or constructor terminates abnormally and throws and exception. Thus, methods and constructors should preserve and establish invariants both in normal and exceptional behaviour. This means that both post-conditions and exceptional post-conditions clauses will implicitly include the invariants. Moreover, under the referred conditions, if a method or constructor violates the invariant in case of abrupt termination, one will typically try to strengthen the pre-condition or weaken the invariant in order to deal with the exceptional behaviour [6]. If a method is pure, automatically preserves the invariants.

In an inheritance perspective, a given class will inherit all the visible invariants from its superclass or superinterfaces. The inherited invariants are composed only by the instance invariants. The static invariants are related to a specific class and should not be passed through.

This notion of static and instance invariants is also present in *VDM++* as it was explained above. However, *JML* invariants have access modifiers which are not present in *VDM++*. Thus, in order to maintain the semantic of such constructs, it will be require at the *VDM++* level to have invariants separated by the access modifiers of the instance variables used for the formulation of properties. For example, the user should not specify an invariant both using public and private variables; such an invariant should be split in two invariants, one related to the public variable and other to the private variable, otherwise, the semantic would be lost. Furthermore, if one at the *VDM++* level define such an invariant, both with public and private variables, when moving such invariant to the *JML* side, it would result in an error.

When there is no invariant defined in a class, the default value is true. On the other hand, if there are more than one invariant, the resulting invariant is a conjunction of all invariants of the class.

Finally, unlike *VDM++*, *JML* does not have type invariants. However, a type in *VDM++* is equivalent to a class or type in *JML*. If the *VDM++* type in question is a pre-built type in *VDM++*, then it corresponds to a type in *JML*. From that fact, results two possibilities:

- Create a new class extending the correspondent type defined in 3.1 with the invariant;
- Instantiate the invariant for each variable that uses the correspondent type or create a subclass of the type being used at the *JML* side containing the invariant and each variable that uses the *VDM++* type will use at the *JML* side the new class created as type.

In detail, if the type is defined by the user as a record, then it corresponds to a class in *JML*. Thus, the associated invariant is semantically equivalent to a class invariant, and can be added to the created class representing the record as an instance invariant. If the type is a pre-built type in *VDM++*, then there are two options. One option is to instantiate the invariant for each variable of the type that contains the invariant. This means that the type invariant will become an instance invariant over the specific variables. However, if there are a number of variables using the specific type, there will be the same number of invariants at the *JML* side. Thus, there will be repeated information in order to preserve the invariant. Other possibility is to extend the current defined types in *JML* to be mapped to *VDM++*. For example, if there is a type at *VDM++* side named *A*:

$$A \quad = \quad \textit{set of nat};$$

It would be mapped to a *JmlValueSet*. However, if a new class is created (*e.g.*, named *JmlValueSetExtended*) as an extension of the *JmlValueSet* (*i.e.*, a subclass of), and the invariant is added to that class, then the type *A* would be mapped to *JML* not as a *JmlValueSet* but as a *JmlValueSetExtended*. Due to the fact that the new type only adds a new invariant imposed by the type invariant at *VDM*++ level, the semantic meaning of the type wil be maintained.

3.2.5 Map *VDM*++ External Clause to *JML* Assignable Clause

The external clauses in *VDM*++ list a number of variables that a given function or operation will manipulate along its execution. With this clause, it is also possible to limit the usage of the variable by choosing to read it only, or read and write. However, the externals clause can only be used within an implicit style.

The assignable clause in *JML* lists the locations the operation can assign during its execution. Thus, the operation can only manipulate the information contained within that name space.

As it can be seen from both definitions, both the externals and the assignable clause are similar, and it is possible to map them in both ways.

When moving from *VDM*++ to *JML*, all the external clauses names will be moved to the correspondent assignable clause, maintaining the semantic value of the construct. On the other hand, when moving from *JML* to *VDM*++, an assignable clause can only be moved into an implicit function or operation. If that happens, the assignable list will be mapped into an externals clause, with the write permissions. This way, the semantic value of both operators is maintained when using them within this mapper.

Chapter 4

Preliminary design

This chapter aim to give a complete description of the preliminary design of the connection between *VDM++* and *JML*. The connection itself should be built under the support of a number of components described in this section, which will allow the interaction of a number of technologies that will support the usage of the proposed connection.

Along this chapter, it is possible to see the overview of the preliminary design of this tool in section 4.1. Furthermore, in section 4.2, the Overture and *JML* parsers are explained, including its interaction with this connection between *VDM++* and *JML*. Finally, the section 4.3 provides an explanation of the *JML* AST developed in *VDM-SL*.

4.1 Overview

In order to create the proposed connection between *VDM++* and *JML*, it is expected that most of its components will be specified in *VDM++* and then use the VDMTools Java code generator to create the correspondent *JAVA* classes from the *VDM++* specification [2]. After this process, the *JAVA* classes can be gathered in an eclipse plugin for further usage.

Conceptually, this will be a bidirectional connection between *VDM++* and *JML*, as illustrated in figures 4.1 and 4.2. It will make use of syntactic analysis in a way that each specification, placed in files, should be analysed syntactically, *i.e.*, parsed into an intermediate structure. This is performed by two components called scanners and parsers. The scanner is a lexical analyser which creates tokens (categorized blocks of text) from a sequence of inputs and is used by the parser in order to transform the input sequence to an Abstract Syntax Tree. However, this AST must be built depending on each construct of the language being parsed. This means that each production in the parser should have the correct

information with respect to the correspondent construct being parsed. This way, each node of the AST will gather information of the construct being parsed. The structure of the parsing component and the associated building process are explained in section 4.2. Concerning the AST definitions, it is presented in section 4.3.

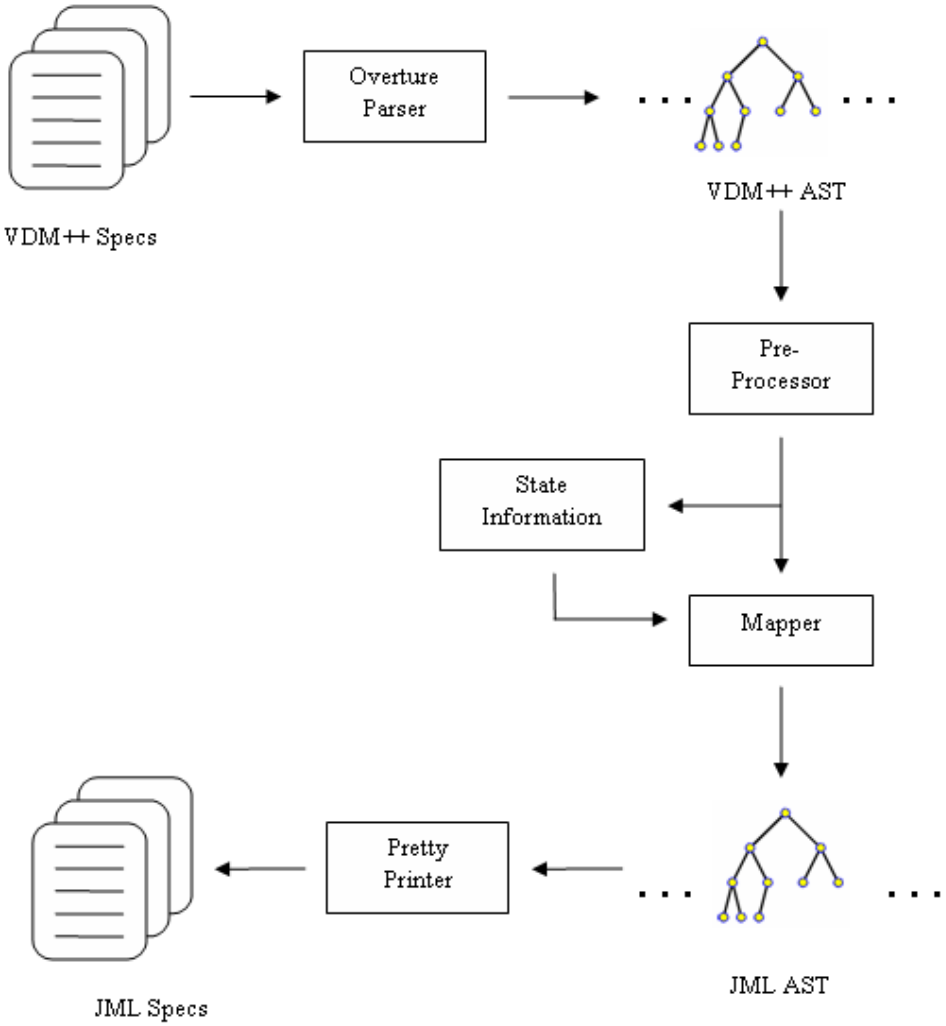

Figure 4.1: Illustration of the overview of this connection from *VDM++* to *JML*.

Afterwards, the corresponding AST structure should be mapped into another AST structure representing the specification language one wants to move to. This means that if one wants to move from *VDM++* to *JML* (or vice-versa), the resulting AST from parsing the *VDM++* (*JML*) file should be mapped into another AST representing *JML* (*VDM++*). This conversion between ASTs is explained in

detail in chapter 5.

The resulting ASTs from the mapper briefly explained above contain all the abstract syntax of the corresponding specification languages, *i.e.*, they contain all the information derived from the concrete syntax of a language. However, in order to have the final output files, those ASTs should be pretty printed to the correspondent syntax of the specification language in question. This means that both *JML* and *VDM++* abstract syntax trees should have operations to pretty-print the abstract syntax to the concrete syntax of the correspondent language. This component will not be explained in this thesis and it should be considered as future work, since it will be developed in *JAVA*.

In order to understand these concepts, the figures 4.1 and 4.2 are showned below. The intention of such images is to provide a full overview of this connection in both ways. Note that the *pretty-printers* were not specified, and should be seen as future work. Furthermore, the figures show how the different components hang themselves together, and the number of different steps in which the specifications are subject to processing.

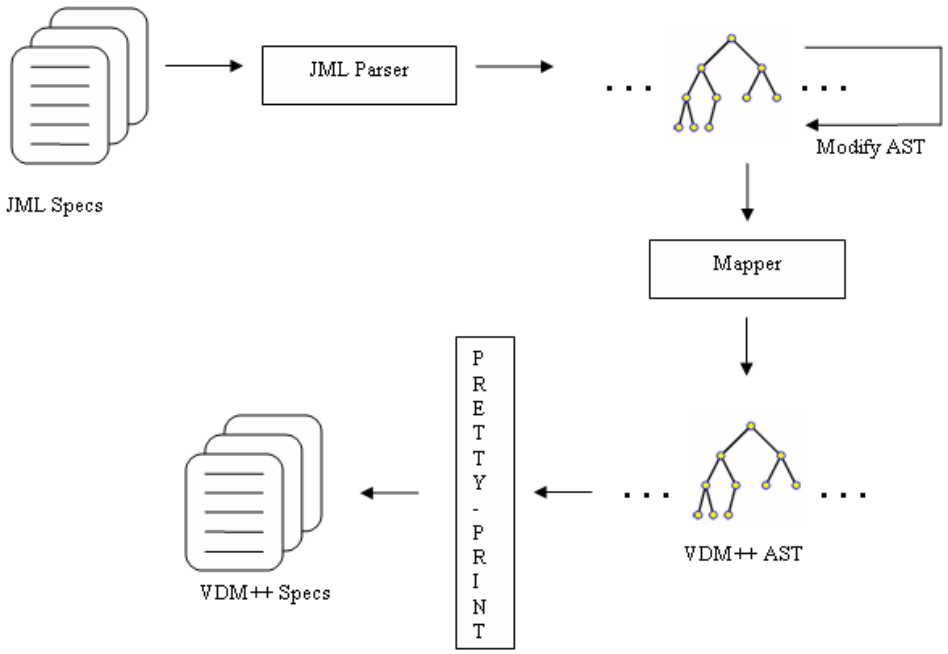

Figure 4.2: Illustration of the overview of this connection from *JML* to *VDM++*.

4.2 Parsers

This component will not be specified, due to the fact that there are parsers already defined for *VDM++* and *JML*. Thus, there is no need to re-implement the corresponding parsers. However, an interaction between the parsers and the proposed connection should be explored in order to define the correct interfaces to perform the communication between them. In the following subsections, an analysis of the correspondent interfaces with the parsers will be carried through in both sides of this connection.

4.2.1 VDM++ Parser

At the *VDM++* side, the Overture parser will be used. This parser allows one to retrieve, from *VDM++* specifications, the relevant information, which are the ASTs. It is possible from a given specification to return ASTs, representing the specification parsed, as *VDM++* values. Those values can then be used for further process. With this functionality, one can give *VDM++* specifications as input to the parser, and retrieve a forest of ASTs representing the abstract syntax of those input files.

Thus, it is possible to parse a number of *VDM++* files and retrieve the corresponding ASTs using the Overture parser. Thus, there is no need to specify this part of the first component.

4.2.2 JML Parser

At the *JML* side, the fourth version of the *JML* tools [1] will be used for the sake of simplicity and compatibility. This version of the JML tools is an Eclipse-based set of tools, which includes a *JML* parser that will be used to parse *JML* input files and build ASTs representing the input files. However, this is not as straightforward process as on the *VDM++* side. This bidirectional connection between *VDM++* and *JML* is intended to be a prototype proof-of-concept connection between subsets of those two specification languages with a solid basis with special attention concerning future extensions of it. Following this principle, the current ASTs retrieved by the *JML* parser cannot be used directly. The presence of information about constructs not considered in this first version of this connection should not exist, to simplify the mapping between the ASTs from *JML* and *VDM++*. Thus, a transformation of the retrieved ASTs should be carried through in order to transform them into ASTs considering only the constructs used. This includes two steps:

- Creation of abstract syntax types representing the considered *JML* types, with special attention to future extensions;

- Converting the nodes of the retrieved ASTs from the *JML* parser to the new constructs created in the previous step.

The first item presented above suggests a creation of abstract syntax types representing the subset of *JML* considered in this connection. This process is explained in detail in section 4.3.

The second step evolves the conversion between ASTs. As it can be seen above, the ASTs retrieved by the *JML* parser contains the complete *JML* constructs from the input files. In case some of those constructs are not considered yet in this connection, they should be ignored to avoid excess of information and complexity when mapping ASTs. Thus, the overall goal of this step is to visit each node of the ASTs and:

- If the construct present on the node is considered in this connection, it will be replaced by the corresponding one generated as a *JAVA* class in the previous steps;

- If the construct is not yet considered in this connection, it will be ignored.

For extension purposes, if one wants to expand the subset of considered constructs in a future stage, one have to:

- Write the abstract syntax of the construct as a *VDM-SL* type inside the abstract syntax file representing *JML* (as suggested in section 4.3);

- In the ASTs converter, one will change the visitor pattern in order to write the correct construct in the right place instead of ignoring it.

In short, in this step the abstract syntax of *JML* will need to be specified. The specifications is shown in section 4.3. Concerning all the other steps explained above, they will be implemented in *JAVA*.

4.3 JML Abstract Syntax Tree

In order to be able to map *JML* and *VDM++*, a connection between the abstract representation of both languages should be preformed. However, it is necessary to have such abstract representation.

Concerning *VDM++*, such representation already exists. Thus, the first step evolves the specification of an abstract representation of *JML*, *i.e.*, a representation with no *syntactic sugar*, by means of *VDM-SL* types. In this representation, only the constructs considered from *JML* will be designed and if ones wants to extend the subset of *JML* it will specify the new constructs in this file, and consider the generated *JAVA* classes in the next item as explained below.

As it was said above, an abstract syntax representation of *VDM++* already exists, defined using *VDM-SL* types. Thus, each construct present in the language has an abstract representation as a *VDM-SL* type. For example, a *VDM++* class can be represented by the following *VDM-SL* type:

```
ExplicitOperation ::
  identifier     : Identifier
  type           : Type
  parameter_list : seq of Pattern
  body           : OperationBody
  trailer        : OperationTrailer
```

As it can be seen from the type presented above, a *VDM++* explicit operation can abstractly be represented by:

- An identifier, which represents the name of the operation;
- A type, which represents the output type of the operation;
- A list of parameters, which represents the input parameters of the operations;
- A body, which can be one of the three possibilities *VDM++* allows, *i.e.*, not yet specified, subclass responsibility or a concrete implementation and;
- finally, the operation trailer, which represents assertions over the operation (*i.e.*, pre-, post-conditions, etc)

The same will occur to all the *VDM++* existing constructs.

Concerning *JML*, the same procedure occur. Thus, a set of *VDM-SL* types was created in order to abstractly represent the language. As an example, the *VDM-SL* types created to represent an operation in *JML* is as follows:

```
OperationDefinition ::
  identifier     : Identifier
  access         : AccessDefinition
  pure           : bool
  statickeyword  : bool
  final          : bool
  returning_type : Type
  parameter_list : seq of Parameter
  body           : [Body]
  trailer        : [MethodSpecifications];
```

An operation in *JML* can be represented by:

- An identifier, which represents the name of the operation;
- An access definition, which represents the visibility of the operation;
- Three keywords represented as boolean values which will represent if a class is pure, static or final;
- A returning type;
- A list of parameters;
- Possibly a body, if it is not an interface; and
- Finally, the trailer, which represents pre-conditions, post-conditions, etc.

As one can see from both types presented above, the representation of both languages is similar and this way a common representation was found in order to perform the mapping between the constructs.

After having this representation in *VDM-SL* types, a number of classes should be generated representing each *VDM-SL* type, with operations over the attributes of the corresponding type. This will allow a clean generation of *JAVA* code and an straightforward mapping between the constructs of both *VDM++* and *JML*. This process evolves the use of a tool designed for the Overture project. The tool is called *ASTGEN* and from *VDM-SL* types representing a language it generates *JAVA* classes and interfaces and *VDM++* classes representing those types. As an example, for the *JML* abstract representation of the pre-condition (*i.e.*, ensures clause):

```
EnsuresClause ::
  ensures_expression : Expression;
```

The following class will be generated:

```
class JmlEnsuresClause is subclass of IJmlEnsuresClause
operations
  public identity: () ==> seq of char
  identity () == return "EnsuresClause";

  public accept: IJmlVisitor ==> ()
  accept (pVisitor) == pVisitor.visitEnsuresClause(self);

  public JmlEnsuresClause:
    (IJmlExpression) ==> JmlEnsuresClause
  JmlEnsuresClause (p1) ==
    ( setEnsuresExpression(p1) );
```

```
  public JmlEnsuresClause:
    (IJmlExpression) *
    nat *
    nat ==> JmlEnsuresClause
  JmlEnsuresClause (p1,line,column) ==
    ( setEnsuresExpression(p1);
      setPosition(line, column) );

  public init: map seq of char to [FieldValue] ==> ()
  init (data) ==
    ( let fname = "ensures_expression" in
        if fname in set dom data
        then setEnsuresExpression(data(fname)) );

instance variables
  private ivEnsuresExpression : [IJmlExpression] := nil

operations
  public getEnsuresExpression: () ==> IJmlExpression
  getEnsuresExpression() == return ivEnsuresExpression;

  public setEnsuresExpression: IJmlExpression ==> ()
  setEnsuresExpression(parg) == ivEnsuresExpression := parg;

end JmlEnsuresClause
```

As it can be seen from the presented class, the class has an instance variable representing the expression of the pre-condition named *ivEnsuresExpression*, getters and setters over that variable, constructors and a number of other helper methods. From those classes (one for each type present in the abstract representation of *VDM++* or *JML*), it is possible to map the two languages by means of a *VDM++* specification, as it can be seen from chapter 5.

The complete representation of *JML* using *VDM-SL* types can be seen in appendix A. All the types are commented in order to understand their meaning.

Chapter 5

Connection Specification

This chapter aim to give a complete description of the behaviour of this connection. Thus, a description of each part of this connection will be given and the interaction with the users will be explored. In order to better understand this connection, it will be decomposed into well defined logical components. Each component will have its own interfaces defined in order to understand how the communication between those components works. Together with the explanation, this connection will also be illustrated through figures of each component, with particular emphasis on the component describing the mapper between *VDM++* and *JML*.

5.1 Overview

After completing the previous steps described in chapter 4, one will have ASTs representing input files. In order to map specifications both from *VDM++* and *JML*, a mapper should be defined. This mapper should be able to convert *VDM++* ASTs into *JML* ASTs and vice-versa. In virtue of the ASTs store all the necessary information in order to convert to other ASTs representing the target language. However, such a conversion should respect the semantic rules defined in chapter 3, in order to maintain the semantic value of the languages.

Such a mapper should allow one to move freely from *VDM++* to *JML* and vice-versa. However, there will be a pre-processor responsible to check if it possible to preform the connection, based on the semantic rules of both languages.

The referred pre-processor is explained in detail in section 5.2. Furthermore, in section 5.3 one can see the specification and corresponding explanation of the state information. Finally, in section 5.4 one can see some of the key operations of the specification of the mapper both from *VDM++* to *JML* and *JML* to *VDM++*. Note that the pre-processor and the state information are only neces-

sary when moving from *VDM++* to *JML*, thus the explanations will be focused on this assumption.

5.2 Pre-processor

The pre-processor component of this specification if of great importance concerning the mapper from *VDM++* to *JML*. As it can be seen from chapter 3, there are three key situations where *VDM++* specifications should be pre-processed:

- If there exists multiple inheritance, it should be resolved because *JML* does not allow multiple inheritance;
- If there is type information in the specification, which is not a composite type, it should be saved into an intermediate structure in order to consult the name of the type at the *JML* side; and
- Whenever a *VDM++* specification has composite type definitions, a new class should be created to each of those composite types.

Each of these three points will be explained in detail below. A number of operations from the *VDM++* specification of the pre-processor are shown to illustrate how these concepts are used in practice.

Starting with the multiple inheritance issue, it is necessary to resolve it when moving from *VDM++* to *JML*. The adopted solution was explained in subsection 2.2.1, and it is illustrated in figure 5.1. In figure 5.1, the parsing component of *VDM++* is omitted. The ASTs will then be pre-processed by the pre-processor. That component will initially check if there is any multiple inheritance present.

In case there are no multiple inheritance, the pre-processor will gather type and class information into a state component, which is explained in section 5.3, and it start the mapper component, explained in section 5.4.

On the other hand, if there is multiple inheritance, the pre-processor will try so resolve it, *i.e.*, it will try to eliminate the multiple inheritance applying the rules explained in section 2.2.1. If it is not possible to satisfy the referred rules, the pre-processor will finish its execution and the mapper will not run because it would generate erroneous classes, with multiple inheritance, which is not supported at the *JML* side.

However, if it is possible to eliminate the multiple inheritance, the pre-processor will eliminate it, gather type and class information and finally it will proceed to the mapping component. Note that the elimination of the multiple inheritance will consist in mapping one or more classes into interfaces. That information will also hold in the state information, as explained in section 5.3. That information will then be used by the mapper for further treatment.

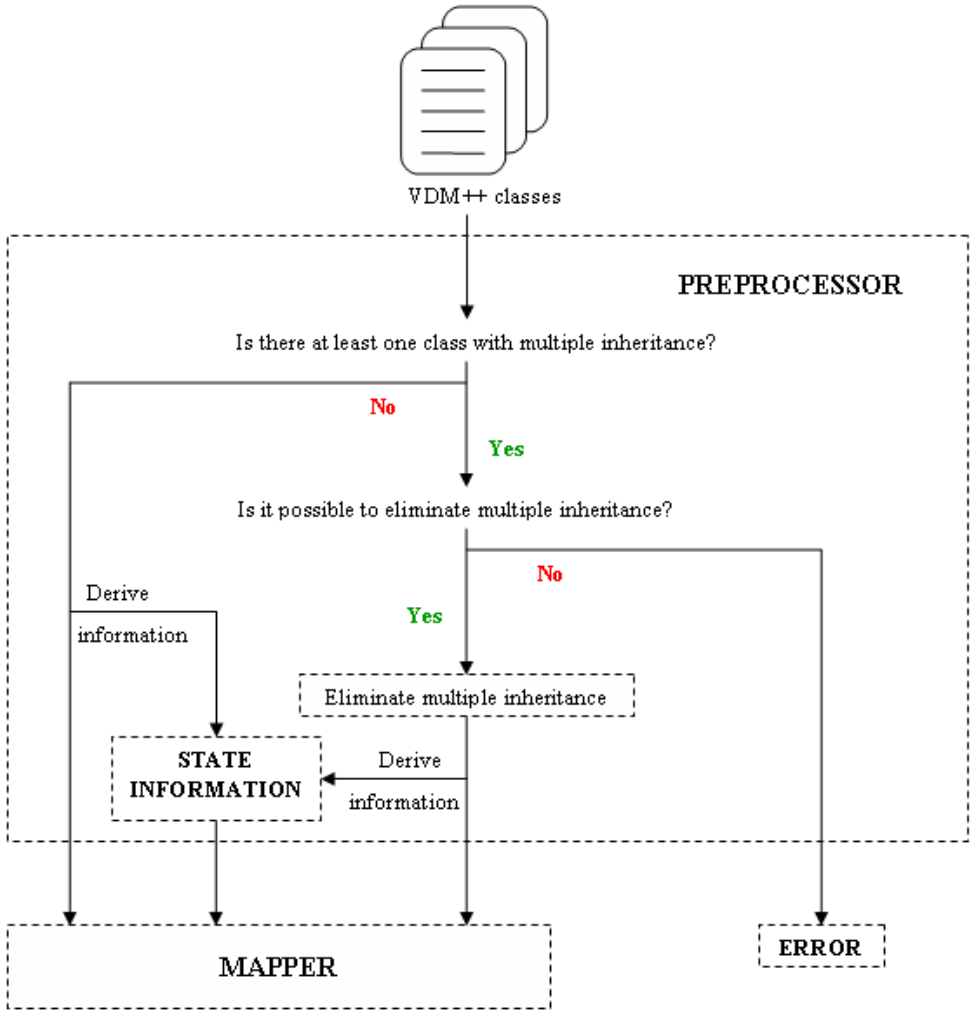

Figure 5.1: Illustration of the pre-processor over VDM++ specifications.

The operations responsible for this are presented below. The operation *init* is the responsible for activating the pre-processor in order to understand if there is any multiple inheritance.

```
public init :
  OmlSpecifications ==>
  JmlSpecifications
init(specs) ==
  if preprocess(specs)
  then return eliminateMI(specs)
  else return build_jml(specs);
```

If the specifications have multiple inheritance, the pre-processor tries to eliminate it through the following operation:

```
public eliminateMI :
  OmlSpecifications ==>
  JmlSpecifications
eliminateMI(specs) ==
  if canProceed(specs)
  then (eliminate(specs);
        build_jml(specs))
  else return new JmlSpecifications([]);
```

This operation will call the operation *canProceed* which will try to see if it is possible to eliminate the multiple inheritance. This is preformed by checking if at most one of the superclasses of the class in question is a class and if all the other superclasses can be transformed into interfaces.

Finally, if it is possible to eliminate the multiple inheritance, the *eliminate* operation will update the state information in order to be used by the mapper. Below, one can see the mentioned operation:

```
public eliminate :
  OmlSpecifications ==>
  ()
eliminate(specs) ==
  (interfaces_list := updateInterfaceList(specs);
   classes_list := updateClassList(specs));
```

The operations that checks if a given class has multiple inheritance is the operation *hasMI*, and its specification is presented below:

```
public hasMI :
  OmlClass ==>
  bool
hasMI(c) ==
  let ic = c.getInheritanceClause()
  in if c.hasInheritanceClause()
     then let l = len ic.getIdentifierList()
               in if l > 1
                        then return true
     else return false
         else return false;
```

It will check if a given class has more than one superclass, and it will return the verdict: *true* if it has mutliple inheritance and *false* if it has no multiple inheritance.

5.3 State Information

In order to be able to preform the mapping between the two specification languages in an efficient way, it is important to gather a number of relevant information before mapping the ASTs. Afterwards, the mapper will then access that state information in order to preform actions such as decide about inheritance and resolve type names.

The state information showed in figure 5.1 is composed by the following elements:

- A set of strings, representing *VDM++* class names that will become classes in *JML*;
- A set of strings, representing *VDM++* class names that will become interfaces in *JML*;
- A mapping, where the domain elements is composed by class names and the range elements is composed again another map with information relative to type information;
- Finally, another map where the domain is composed by class names and the range is composed by type information (complex *VDM++* types) that will be transformed into classes in *JML*.

The components mentioned are instance variables that will hold a number of information that will be supplied to the mapper during the transformations. The *VDM++* types can be seen below.

```
types

public Information ::
  field_list : seq of JmlField
  invariant  : [JmlExpression];
public TypeInfo = map ClassId to JmlType;
public ClassId = seq of char;
```

With respect to the instance variables, the *VDM++* specification of then is showed below:

```
instance variables

public to_class : map ClassId to Information;
public hold_type_info : map ClassId to TypeInfo;
public interfaces_list : set of ClassId;
public classes_list : set of ClassId;
```

The first two items, named *interfaces_list* and *classes_list*, represents sequences of class names that will become classes or interfaces in *JML*. Those structures will hold the information gathered by the pre-processor about multiple inheritance. Thus, the pre-processor will first make sure if it is possible to solve the multiple inheritance, and afterwards, if it is possible, it will fill those sequences with the correct information promoting the necessary classes to interfaces and maintaining the others as classes. Whenever the mapper needs to update the information about inheritance of a given class, it will consult the two sets in order to adjust the information correctly.

The operation responsible for updating this information is the operation *eliminate* presented in section 5.2.

The third item showed above, named *hold_type_info*, is a map from *VDM*++ class names to type information. In *VDM*++ the creation of simple types, which are types with an identifier and a type based on the pre-defined types in *VDM*++, is common. One example of those types can be found below:

```
types

public String = seq of char;
```

In this example, the type *String* is created by means of a pre-defined type in *VDM*++which is a sequence of characters. Afterwards, if one uses the created type inside, for example, an operation like the following:

```
operations

public concat : String * String ==> String
concat(s1,s2) == is not yet specified;
```

Then, the mapper needs to know what is the *VDM*++ type behind the keyword *String*, in order to make the transformations to a *JML* type. To preform this, the pre-processor will gather this kind of information. Each class named will be mapped into another map, where the key values are type names (in this example, *String*) and the range will be the corresponding *VDM*++ type already transformed into a *JML* type. In this case, because the *seq of char* is equivalent to the *JML* type *JMLValueSequence*, the *JML* type will be placed as the range of the *String*. Thus, whenever the type name *String* is used, the mapper will consult the mapping in order to obtain the type behind the keyword that identifies it.

Finally, with respect to the fourth instance variable named *to_class*, whenever a class has complex types, they should be mapped into *JML* classes due to the fact they have fields and possible invariants. If there is a type with an invariant such as:

```
types

public PairOfNat ::
  fst : nat
  snd : nat
inv t == t.fst < t.snd;
```

Then, the final map will contain, for the class in question, the information regarding the type *PairOfNat*, where the type name will be the domain and in the range there will be a list of fields and possibly an invariant. In this case, the list of fields will hold the information regarding the fields *fst* and *snd* and the type information associated, already converted to the correspondent *JML* types. Besides that, because the type has an invariant associated, its expression will be transformed into an equivalent *JML* expression representing the invariant, and that information will be saved together with the field list. Note that the operation returns always a *true* value, because it is being used inside a sequence comprehension, which requires the operation to return.

```
public buildClassEntry :
  OmlClass ==>
  bool
buildClassEntry(c) ==
  let cn = c.getIdentifier(),
      bl = c.getClassBody(),
      ts = buildTypeInfo(bl),
      cl = toClassInformation(bl)
  in (
        hold_type_info := hold_type_info ++ { cn |-> ts };
        to_class := to_class ++ cl;
        return true;
     );
```

After the pre-processor gathers all the relevant information, the mapper is ready to convert the two specifications languages. Afterwards, if there are type information that should be mapped into classes, it should be performed by a post-processor over the type information gathered in the instance variable *to_class*.

5.4 Mapper

After performing the actions described in sections 5.2 and 5.3, the mapper is ready to be used. The goal of this mapper is to convert ASTs from *JML* to *VDM++* and vice-versa. As it was said above, the pre-processor will only be

needed when moving from *VDM++* to *JML*. Besides this, the two mappers are very similar, and their goal is to map a number of constructs using the abstract syntax defined in *VDM-SL*, as it can be seen from appendix A. A number of relevant operations that perform such transformations is presented in the subsections below. Figure 5.2 illustrates the mapper functionality, and as it can be seen, it suggests a link comming from the pre-processor figure 5.1.

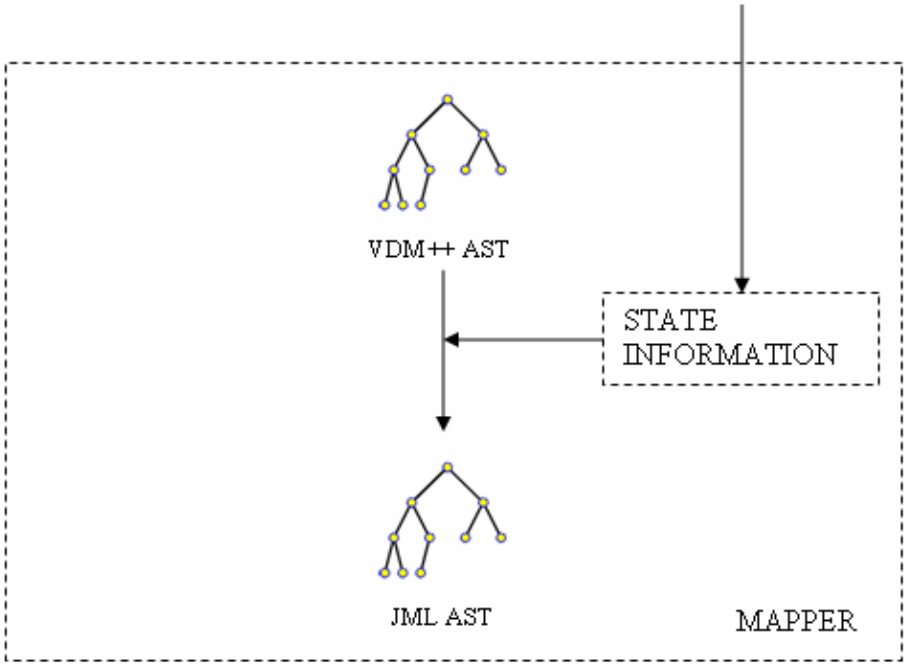

Figure 5.2: Illustration of the mapper functionality.

5.4.1 Mapper from *VDM++* to *JML*

The main operation of this mapper, called *build_jml*, is the responsible for mapping each class present in the AST into *JML* classes.

```
public build_jml :
  OmlSpecifications ==>
  JmlSpecifications
build_jml(specs) ==
  let classes = specs.getClassList(),
      jml_classes = convertVdm2JmlClasses(classes)
  in return new JmlSpecifications(jml_classes);
```

As it can be seen, the operations transforms each class into a *JML* class. The operation responsible for mapping a *VDM++* class into a *JML* class is the following:

```
 public convertVdm2JmlClass :
  OmlClass ==>
  JmlWrappedJmlClass
convertVdm2JmlClass(c) ==
  let id   = c.getIdentifier(),
          inh  = c.getInheritanceClause(),
      body = c.getClassBody(),
      jmlclass = build_class(id, inh, body),
      jmlimp = getJmlImports(),
      javaimp = getJavaImports()
  in return new JmlWrappedJmlClass([],[],javaimp,jmlimp,jmlclass);
```

The subsequent operations are similar, and the goal is to map construct by construct, respecting the rules already defined and the limitations encountered. From those operations, there is a number of them responsible for the main blocks present in a *VDM++* specification.

The values block inside a *VDM++* specification is mapped into a similar values block in *JML*, which represents static variables inside the code. Thus, each value in a *VDM++* specification is mapped into a static variable in *JML* by means of the operation *buildJmlValue*, presented below:

```
public buildJmlValue :
  OmlValueDefinition ==>
  JmlValueDefinition
buildJmlValue(val) ==
  let access = val.getAccess(),
      shape = val.getShape(),
      newAccess = buildJmlAccess(access),
      newShape = buildJmlShape(shape)
  in return new JmlValueDefinition(newAccess,true,true,newShape);
```

Furthermore, each instance variable in a *VDM++* class is mapped into a *JML* variable by means of the following operation:

```
public buildVariables :
  OmlInstanceVariable ==>
  JmlVariable
buildVariables(var) ==
  let access = var.getAccess(),
      scope = access.getScope(),
      assign = var.getAssignmentDefinition(),
```

```
      newScope = new JmlScope(scope.getValue()),
      newAccess = new JmlAccessDefinition(newScope),
      stat = access.getStaticAccess(),
      tp = convertType(assign.getType()),
      id = assign.getIdentifier(),
      expr = convertExpression(assign.getExpression())
  in return new JmlVariable(newAccess, true, stat,
                            false, tp, id, expr);
```

This operation will gather all the information necessary to convert a *VDM++* instance variable into a *JML* variable. Moreover, each operation in *VDM++* is mapped into a *JML* operation using one of three operations, because it is possible to have three kinds of operations in *VDM++*: implicit, explicit and extended explicit. As an example, the operation that converts an explicit *VDM++* operation into a *JML* operation is showed below:

```
public buildExplicitJmlOperation :
  JmlAccessDefinition *
  bool *
  OmlExplicitOperation ==>
  JmlOperationDefinition
buildExplicitJmlOperation(access,statickey,shape) ==
  let id = shape.getIdentifier(),
      tp = shape.getType(),
      type_rng = getRngType(tp),
      params = shape.getParameterList(),
      trailer = buildOperationTrailer(shape.getTrailer(),access),
      paramsList = buildOperationParameters(params,tp)
  in return new JmlOperationDefinition(trailer,access,true,
     statickey,false,type_rng,id,paramsList,nil);
```

Besides the operation, all its trailers, *i.e.*, pre-, post-conditions, error and external clauses are also mapped by means of the operation *buildOperationTrailer*, provided in appendix B.

With respect to types, they are mapped using operations such as the following one, that converts a *VDM++* product type into a *JML* tuple type, created to hold this kind of information.

```
public convertProductType :
  OmlProductType ==>
  JmlTupleType
convertProductType(t) ==
  let tp = buildSeqTypes(t),
      sq = [ convertType(tp(i)) | i in set inds tp ]
  in return new JmlTupleType(sq);
```

Any other *VDM++* block used in a *VDM++* class will be ignored, and thus it will not be mapped.

Although the following conversions are not between blocks, they are essential to the entire mapping. This conversions are between expressions and literals. Expressions are present in every sentence of a language, and they are connected by means of operations such the following, that converts a *JML let* expression into a definition block in *JML*:

```
public convertLetExpression :
  OmlLetExpression ==>
  JmlBlockExpression
convertLetExpression(e) ==
  let bind = e.getDefinitionList(),
      newbind = buildJmlShapes(bind),
      expr = e.getExpression(),
      res = convertExpression(expr)
  in return new JmlBlockExpression(newbind,res);
```

There are a number of similar operations responsible for converting a number of *VDM++* expressions into *JML* expressions, which are listed and explained in appendix B.

Finally, literals allow one use elements of a give type. One example of an operation is as follows:

```
public convertBooleanLiteral :
  OmlBooleanLiteral ==>
  JmlBooleanLiteral
convertBooleanLiteral(n) ==
  let val = n.getVal()
  in return new JmlBooleanLiteral(val);
```

The complete specification and description is presented in appendix B. The limitations that this mapper imposes were defined in chapter 3. After performing the pre-processing stage and after running the mapper over *VDM++* ASTs, it is possible to have *JML* ASTs ready to be pretty-printed into concrete classes. Examples of this mapper and the pre-processor can be found in chapter 6. Two case studies were used in order to test the functionalities explained along this chapter.

5.4.2 Mapper from *JML* to *VDM++*

This mapper from *JML* to *VDM++* is similar to the one mentioned in subsection 5.4.1, with respect to the mapping of the ASTs. Concerning the pre-processing

part, it is not required here because there are no limitations excepting the usage of constructs that are not being considered by this connection.

Thus, the main operation of this mapper is the operation *build_vdm*, presented below.

```
public build_vdm :
  JmlSpecifications ==>
  OmlSpecifications
build_vdm(specs) ==
  let classes = specs.getClassList(),
      vdmclasses = convertJmlClasses(classes)
  in return new OmlSpecifications(vdmclasses);
```

This operation will transform each *JML* specification received as input into a correspondent specification in *VDM++*. Each specification has a class list, and the operation that transforms each *JML* class into a *VDM++* class is the following:

```
public convertJmlClass :
  JmlWrappedJmlClass ==>
  OmlClass
convertJmlClass(c) ==
  let cl = c.getClassVal(),
      id = cl.getIdentifier(),
      ic = cl.getInheritanceClause(),
      ii = cl.getInterfaceInheritance(),
      ih = getInheritanceClauses(ic,ii),
      bd = cl.getClassBody(),
      body = convertClassBody(bd)
  in return new OmlClass(id,[],ih,body,false);
```

Further on, it is necessary to transform each *JML* block into a *VDM++* block (e.g., operations block). Below, a number of operations responsible for the transformation of each considered block is presented and explained.

Starting with the operation named *convertInstanceVariables*, it converts *JML* instance variables into *VDM++* instance variables. The other operations in use are listed and explained in appendix B.

```
public convertInstanceVariables :
  JmlInstanceVariableDefinitions ==>
  OmlInstanceVariableDefinitions
convertInstanceVariables(s) ==
  let jml_vars = s.getJmlVariables(),
      java_vars = s.getJavaVariables(),
          vdm_1 = convertVariables(jml_vars),
```

```
        vdm_2 = convertVariables(java_vars),
        res = vdm_1 ^ vdm_2
  in return new OmlInstanceVariableDefinitions(res);
```

Furthermore, the operation *convertValueDefinitions* converts *JML* values into *VDM++* values. The specification of the referred operation is as follows:

```
public convertValueDefinitions :
  JmlValueDefinitions ==>
  OmlValueDefinitions
convertValueDefinitions(s) ==
  let l = s.getValueList(),
      q = convertValues(l)
  in return new OmlValueDefinitions(q);
```

The invariant definition present in *JML* are mapped into *VDM++* invariants applied to instance variables, *i.e.*, they are written as instance variables invariants within the instance variables block. The operation responsible for such transformation is shown below.

```
public convertInvariantDefinitions :
  JmlInvariantDefinitions ==>
  OmlInstanceVariableDefinitions
convertInvariantDefinitions(s) ==
  let l = s.getInvariantList(),
      q = convertInvariants(l)
  in return new OmlInstanceVariableDefinitions(q);
```

With respect to operations, *JML* operations are converted into *VDM++* implicit operations for two reasons:

- This connection does not connect methods implementations, thus there is no need to have explicit operations;
- If a *JML* operation has an assignable clause, which is equivalent to the *VDM++* external clause, it can only be mapped if the *VDM++* operation is implicit, because explicit operations does not allow the usage of such clause.

Thus, the operation responsible for converting a *JML* operation into a *VDM++* implicit operation is the operation called *convertOperation*.

```
public convertOperation :
  JmlOperationDefinition ==>
```

```
  IOmlOperationDefinition
convertOperation(op) ==
  let access = op.getAccess(),
      stat = op.getStatickeyword(),
      newaccess = buildAccessDefinition(access,stat),
      id = op.getIdentifier(),
      t = op.getReturningType(),
      p = op.getParameterList(),
      tp = buildOperationType(t,p,id),
      params = buildParametersList(p),
      trl = op.getTrailer(),
      trailer = buildOperationTrailers(trl),
      shape = new OmlImplicitOperation(id,params,tp,trailer)
  in return new OmlOperationDefinition(newaccess,shape);
```

There are no other blocks being considered in this mapping from *JML* to *VDM++*. However, it is still necessary to map types, expressions and literals.

Starting with types, each *JML* type that has a correspondence at the *VDM++* level is being mapped by means of operations such as the following operation, which converts a *JML* map type to a *VDM++* map type.

```
public convertMapType :
  JmlMapValueToValueType ==>
  OmlGeneralMapType
convertMapType(m) ==
  let domtp = m.getDomType(),
      rngtp = m.getRngType(),
          tpd = convertType(domtp),
          tpr = convertType(rngtp)
  in return new OmlGeneralMapType(tpd,tpr);
```

The other operations responsible for the conversion of the rest of the types are present in appendix B.

With respect to expressions, an example of converting a *JML* expression into a *VDM++* expression can be found below.

```
public convertBinaryExpression :
  JmlBinaryExpression ==>
  OmlBinaryExpression
convertBinaryExpression(e) ==
  let lhs = e.getLhsExpression(),
          op  = e.getOperator(),
          rhs = e.getRhsExpression(),
          nlhs = convertExpression(lhs),
          nop = convertBinaryOperator(op),
          nrhs = convertExpression(rhs)
  in return new OmlBinaryExpression(nlhs,nop,nrhs);
```

The operation above converts a *JML* binary expression into a *VDM++* binary expression, by means of other operations presented in appendix B.

Finally, the operation *convertLiteral* is responsible to infer the input type in order to assign the correspondent operation that will convert the literal expression.

```
public convertLiteral :
  IJmlLiteral ==>
  IOmlLiteral
convertLiteral(lit) ==
  cases true:
    (isofclass(JmlNumericalLiteral,lit))
          -> return convertNumericalLiteral(lit),
    (isofclass(JmlFloatLiteral,lit))
          -> return convertFloatLiteral(lit),
    (isofclass(JmlEnumLiteral,lit))
          -> return convertEnumLiteral(lit),
    (isofclass(JmlBooleanLiteral,lit))
          -> return convertBooleanLiteral(lit),
    (isofclass(JmlCharacterLiteral,lit))
          -> return convertCharacterLiteral(lit),
    (isofclass(JmlStringLiteral,lit))
          -> return convertStringLiteral(lit),
        (isofclass(JmlNullLiteral,lit))
          -> return new OmlNilLiteral(),
        others
          -> return new OmlNilLiteral()
  end;
```

From this chapter, it was possible to see the procedures of the defined maped, and a number of operations responsible for converting between the two specification languages in question. More information about the specification in *VDM++* of this mapper can be found in appendix B.

Chapter 6

Case Studies

Along this chapter, the used case studies are explained in detail. In section 6.1, an overview is given. Furthermore, in section 6.2, the Alarm System case study is presented, together with its results. Finally, in section 6.3, the second case study, Input/Output Streams, is presented together with its results.

6.1 Overview

The overall intention of this case studies is to show the capabilities of the mapper specified and presented in chapter 5. The first example, the Alarm System, is intended to show the overall functionalities of the mapper. On the other hand, the Input/Output Streams case study is intended to show an example of how the mapper deals with multiple inheritance situations.

6.2 Alarm System

In this section, the Alarm System case study will be presented. This case study, which can be seen in detail in [4], is intended to test the overall functionalities of this mapper.

First, an introduction to the problem will be performed. There are three entities in this Alarm System of a chemical plant: the Expert, the Alarm and the Plant. In this alarm system, whenever an alarm is raised in a plant, an expert with the right qualifications is called to deal with situation. The implementation in *VDM++* of each entity is provided below.

Expert

```
class Expert
```

```
instance variables

quali : set of Qualification;

inv quali <> {};

types
public Qualification =
     <Mech> | <Chem> | <Bio> | <Elec>;

operations
public Expert: set of Qualification
             ==> Expert
Expert(qs) == quali := qs
pre qs <> {};

public GetQuali: ()
          ==> set of Qualification
GetQuali() == return quali;

end Expert
```

Alarm

```
class Alarm

types

public String = seq of char;

instance variables

descr    : String;
reqQuali : Expert‘Qualification;

operations

public Alarm: Expert‘Qualification * String ==> Alarm
Alarm(quali,str) ==
( descr := str;
  reqQuali := quali
);

public GetReqQuali: () ==> Expert‘Qualification
GetReqQuali() ==
  return reqQuali;

end Alarm
```

Plant

```
class Plant

instance variables

alarms   : set of Alarm;
schedule : map Period to set of Expert;
inv PlantInv(alarms,schedule);

functions

PlantInv: set of Alarm * map Period to set of Expert ->
          bool
PlantInv(as,sch) ==
  (forall p in set dom sch & sch(p) <> {}) and
  (forall a in set as &
     forall p in set dom sch &
       exists expert in set sch(p) &
         a.GetReqQuali() in set expert.GetQuali());

types

public Period = token;

operations

public ExpertToPage: Alarm * Period ==> Expert
ExpertToPage(a, p) ==
  let expert in set schedule(p) be st
      a.GetReqQuali() in set expert.GetQuali()
  in
    return expert
pre a in set alarms and
    p in set dom schedule
post let expert = RESULT
     in
       expert in set schedule(p) and
       a.GetReqQuali() in set expert.GetQuali();

public NumberOfExperts: Period ==> nat
NumberOfExperts(p) ==
  return card schedule(p)
pre p in set dom schedule;

public ExpertIsOnDuty: Expert ==> set of Period
ExpertIsOnDuty(ex) ==
  return {p | p in set dom schedule &
              ex in set schedule(p)};

public Plant: set of Alarm *
              map Period to set of Expert ==> Plant
```

```
Plant(als,sch) ==
( alarms := als;
  schedule := sch
)
pre PlantInv(als,sch);

end Plant
```

After running the mapper from *VDM++* to *JML*, the following results are achieved:

Expert The expert specification converted to *JML* is as follows. The type presented is not being considered by the mapper, thus it has no correspondence.

```
//@ model import org.jmlspecs.models.JMLValueSet;
import java.util.Set;

public class Expert{

        //@ public model JMLValueSet quali;

        /*@ public invariant
          @   !quali.isEmpty();
          @*/

        //@ requires !q.isEmpty();
        public Expert(Set q);

        public Set getQuali();

}
```

Alarm This class was mapped without loose of information (except methods bodies, which are not being considered).

```
//@ model import org.jmlspecs.models.JMLString;
//@ model import org.jmlspecs.models.JMLEnumeration;

public class Alarm {

        //@ public model JMLString descr;
        //@ public model JMLEnumeration reqQuali;

        public Alarm(Enum reqQuali, String descr);

        public Enum getReqQuali();
}
```

Plant Finally, the plant class was mapped into *JML*. Because it has a function, which is not being considered in this mapper, the function and where the function is being called has not been mapped. Thus, if one wants to have that specific function being mapped, the boolean expression of its body should be added inside, for example, the instance variables block, as an invariant.

```
//@ model import org.jmlspecs.models.*;
import java.util.*;

public class Plant {

    //@ public model JMLValueSet alarms;
        //@ public model JMLValueToValueMap schedule;


        public Plant(Set s,HashMap m);

        /*@ public normal_behaviour
          @
          @ requires
          @ this.alarms.contains(a) &&
          @ this.schedule.containsKey(p);
          @
          @ ensures
          @ ((Set) this.schedule.get(p)).contains(\result)
          @ &&
          @ \result.getQuali().contains(a.getReqQuali());
          @ assignable this.alarms, this.schedule;
          @*/
        public Expert expertToPage(Alarm a, Object p);


        public Set expertIsOnDuty(Expert p);

        /*@ public normal_behaviour
          @ requires
          @ (this.schedule.keySet()).contains(p);
          @*/
        public int numberOfExperts(Object p);
}
```

The results from this case study were correct, and the mapper showed no problems performing the translation. However, in the case of the Plant class, some constraints were not respected, thus the result is not quite what it was expected.

6.3 Input/Output Streams

Along this section, it is possible to see an introduction to the case study, in subsection 6.3.1, which consists in a description of the problem and the motivation for using it. Furthermore, the specification of the classes can be found in subsection 6.3.2 and finally in subsection 6.3.3, the results of using the bidirectional mapper with this case study are presented along with an analysis of those results.

6.3.1 Introduction

This case study consists in having *VDM++* classes that represents input/output streams. Streams are channels that allow programs to interact with the surrounding environment. This case study was chosen in order to have a multiple inheritance scenario tested by the pre-processor, which is explained in section 5.2, and by the mapper itself, explained in section 5.4. The referred multiple inheritance scenario, called *diamond problem* in object-oriented languages, has to be resolved in order to map the case study from *VDM++* to *JML*.

There are four classes, specified in *VDM++* and presented in section 6.3.2, evolved in this case study: *IO*, *InputStream*, *OutputStream* and *IOStream*. Figure 6.1 illustrates a simple class diagram containing those four classes and how they hand together.

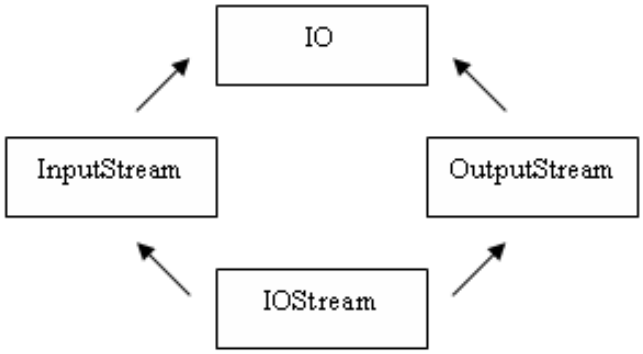

Figure 6.1: Illustration of the input/output stream case study.

As it can be seen from the class diagram, the class *IO* is subclass of both *InputStream* and *OutputStream* classes. Furthermore, the class *IOStream* is subclass of both *InputStream* and *OutputStream* classes. Thus, a multiple inheritance problem arises from this last fact. Note that the methods are not implemented in all the classes and there are no specifications (e.g., pre-conditions)

in any of the classes, because the goal of this case study is to verify how the multiple inheritance issue is solved when moving from *VDM++* to *JML*.

It is expected, using the mapper from *VDM++* to *JML*, that the multiple inheritance issue is detected and resolved by the pre-processor, and also that the mapper will be able to generate the *JML* representation of these classes.

6.3.2 Specification

As it was said in section 6.3.1, this case study is composed by four classes specified in *VDM++*.In this section, the four classes are presented together with a description of each one.

IO The following class is the superclass of both *InputStream* and *OutputStream* classes. It defines the skeleton of any class who implements a stream. The specification of this class can be foune below.

```
class IO

types

public byte = nat;

operations

public read : seq of byte  ==> nat
read(b) == is subclass responsibility;

public write : seq of byte  ==> ()
write(b) == is subclass responsibility;

public close : () ==> ()
close() == is subclass responsibility;

end IO
```

InputStream This class is a subclass of the *IO* class, which represents an input stream. Below, one can find the specification of this class.

```
class InputStream is subclass of IO

operations

public read : seq of byte ==> nat
read(b) == is not yet specified;

public close : () ==> ()
```

```
close() == is not yet specified;


end InputStream
```

OutputStream This class represents an output stream, and its a subclass of the class *IO*, presented above. The specification of this class can be found below.

```
class OutputStream is subclass of IO

operations

public write : seq of byte ==> ()
write(b) == is not yet specified;

public close : () ==> ()
close() == is not yet specified;

end OutputStream
```

IOStream Finally, this class represents a bidirectional stream, both for input and output. It inherits both from the *InputStream* and *OutputStream*, thus this class enforces a multiple inheritance scenario. It contains three methods: *read*, *write* and *close*. The specification of this class is shown below.

```
class IOStream is subclass of InputStream, OutputStream

operations

public read : seq of byte ==> nat
read(b) == is not yet specified;

public write : seq of byte  ==> ()
write(b) == is not yet specified;

public close : () ==> ()
close() == is not yet specified;

end IOStream
```

6.3.3 Results

As it was said in section 6.3.1, the main goal of this case study is to verify how the mapper deals with multiple inheritance situations. Thus, an explanation will be presented below concerning the steps of the pre-processor with respect to these specifications.

The pre-processor will start by checking if there are multiple inheritance within the specification. The main operation *init* presented in chapter 5 will call the operation *preprocess*, which will perform such test. The operation is listed below.

```
public preprocess :
  OmlSpecifications ==>
  bool
preprocess(specs) ==
  let l = specs.getClassList()
  in return hasMultipleInheritance(l);
```

In this case, after executing the *preprocess* operation, the VDMTools interpreter will return the following result:

```
>> p m1.preprocess(v.diamondproblem)
true
```

The operation returned the *true* value, which means that at least one of the classes has multiple inheritance. At this point, the main operation *init* will try to overcome the multiple inheritance problem by invoking the operation *eliminateMI*. That operation is listed below.

```
public eliminateMI :
  OmlSpecifications ==>
  JmlSpecifications
eliminateMI(specs) ==
  if canProceed(specs)
  then (eliminate(specs);
        build_jml(specs))
  else return new JmlSpecifications([]);
```

As it can be seen from the specification, the first test being performed is if it is possible proceed, *i.e.*, if the multiple inheritance can be overcome. The operation responsible for such test is called *canProceed* and it is shown below.

```
public canProceed :
  OmlSpecifications ==>
  bool
canProceed(specs) ==
  let cl = specs.getClassList(),
      s  = gatherInfo(cl)
  in result(s);
```

After executing that particular operation, the results from the VDMTools interpreter are as follows:

```
>> p m1.canProceed(v.diamondproblem)
true
```

Again, the result was positive, *i.e.*, it is possible to eliminate the multiple inheritance issue. Thus, there is at most one class in the inheritance list that can remain a class, and all the other can be promoted to interfaces. In the case of this case study, the two classes in the inheritance list, *InputStream* and *OutputStream* can be promoted to interfaces.

After that, the operation *eliminateMI* can proceed with the elimination of the multiple inheritance. This elimination is gathering the information of which classes can be promoted to interfaces and which classes must remain classes. Furthermore, the *JML* ASTs are built from the *VDM++* ASTs, and the result is an list of *JML* classes. Because there is no pretty-printer available yet, the result cannot be shown . However, after generating the *JML* classes, it is possible to see the state of the inheritance clauses of the *JML* class *IOStream*, which is the only one that had multiple inheritance at the *VDM++* level.

The results are as follows:

```
>> p let jml = m1.init(v.diamondproblem)
     in jml.getClassList()(4).getClassVal().getInterfaceInheritance()
objref2742(JmlInterfaceInheritanceClause):
  < IJmlNode‘ivColumn = 0,
    IJmlNode‘ivInfo = { |-> },
    IJmlNode‘ivLine = 0,
    JmlInterfaceInheritanceClause‘ivIdentifierList = [ "InputStream",
      "OutputStream" ] >
```

From the execution shown above, one can see that the interface list of the class *IOStream* is composed by the two classes that were superclasses at the *VDM++* level. Concerning the class inheritance list, its empty as it can be seen from the execution below:

```
>> p let jml = m1.init(v.diamondproblem)
     in jml.getClassList()(4).getClassVal().getInheritanceClause()
nil
```

Thus, the pre-processor together with the mapper had successfully eliminate the multiple inheritance issue present at the *VDM++* level and had generated correctly *JML* classes. However, without a pretty-printer of the *JML* AST, it is a hard task to analyse the results returned by the VDMTools interpreter. Thus, it is recommended as a future work to develop this component in order to see the results in a more readable perspective.

Using the mapper on the other way arround, *i.e.*, from *JML* to *VDM++*, the same *VDM++* representation was achieved, this the results are not shown, due to the fact that they are identical to the initial classes.

Chapter 7

Conclusion

Along this chapter, the conclusion will be presented. Besides the achievements showed in section 7.1, a number of suggested items as future work is presented in section 7.2.

7.1 Achievements

There were two main goals at the beginning of this project, which were exploring the subsets where a connection between *VDM++* and *JML* was possible and developing a prototype proof-of-concept of such connection. Thus, both goals were achieved at the end of this project. It was possible to find subsets of both languages, explore the limitations within that subset and build the specification of the bidirectional mapper between *VDM++* and *JML*.

Although the mapper is specified, there is work to be done in extending and upgrading it, as suggested in section 7.2, thus this work should be seen as a starting point for extending this connection.

Due to time limitations, it was not possible to present related work. Even though, it would be of interest explore the possibilities of connecting *VDM++* with *SpecSharp*, which is a specification language for the programming language C Sharp.

7.2 Future Work

Although the amount of work accomplished concerning this connection between *VDM++* and *JML*, there are a number of items to finish and to build in order to improve the quality of this tool. Thus, it is suggested to extend the mapper with the following items:

- In order to be able to use this tool in the scope of the Overture Project, it is necessary to code generate the specification of the mapper into *JAVA* and assemble the sources in an Eclipse plugin;
- The pretty-printers of the ASTs both from *VDM++* and *JML* should be developed in order to be able to easily understand the output retrieved by the mapper. This should be performed in *JAVA*, due to its simplicity;
- In order to be able to connect the *JML* parser with this connection, it is necessary to build a visitor pattern over the *JML* AST retrieved by the *JML* parser in order to be able to have the AST converted into the AST presented in this thesis (appendix A);
- After developing an Eclipse plugin of the tool, a Graphical User Interface should be developed in order to ease the interaction of the tool by its users;
- At the specification level, it should be specified a component responsible for gathering information about :
  - potential constructs being used that are not being considered by the mapper;
  - potential semantic losses when moving from one side to another.

  It should be possible to inform the user, by means of a log file, about the situations referred above.
- The considered subset of both *JML* and *VDM++* should be extended in order to allow more features to be mapped in both directions. Thus, the theoretical exploration made along this thesis should be extended, and possibly improved in order to give precise information to the user of this mapper;
- A connection between this mapper and a *JAVA* code generator would be advantageous in a way that should be possible to have concrete *JAVA* implementations being connected to *JML* specifications;
- An explanation of the possible expressions in this map should be provided, and the constructs associated to each type. This would be an advantage to the possible users of this mapper.

Bibliography

[1] Patrice Chalin, Perry R. James, and George Karabotsos. The architecture of jml4, a proposed integrated verification environment for jml. Technical report, Concordia University, Department of Computer Science and Software Engineering, May 2007. [cited at p. 34]

[2] CSK Corporation. *The VDM++ Language*. CSK, 2005. [cited at p. 2, 14, 31]

[3] CSK. VDM homepage. *http://www.csk.com/support_e/vdm/index.html*, 2005. [cited at p. 2]

[4] 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. 2, 8, 55]

[5] A. L. Baker G. T. Leavens and C. Ruby. Preliminary design of jml: A behavioral interface specification language for java. Technical report, Iowa State University, Department of Computer Science, Iowa State University, 226 Atanasoff Hall, Ames, Iowa 50011-1041, USA, May 2003. [cited at p. 2, 11]

[6] et al Gary T. Leavens. Jml reference manual. Available from `http://www.jmlspecs.org`, October 2007. [cited at p. 2, 3, 7, 14, 15, 16, 24, 25, 26, 27]

[7] The VDM Tool Group. The IFAD VDM++ Language. Technical report, IFAD, April 2001.
ftp://ftp.ifad.dk /pub/vdmtools/doc/langmanpp_letter.pdf. [cited at p. 9]

[8] O. Jones. *Introduction to the X Window System*. Prentice-Hall International, 1986. [cited at p. 2]

[9] M. Leino K. Rustan and Peter Muller. A verification methodology for model fields. *Lecture Notes in Computer Science*, Volume 3924/2006:115–130, 2006. [cited at p. 3]

[10] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Jml: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999. [cited at p. 2, 7]

[11] Gary T. Leavens, Yoonsik Cheon, and David R. Cok. Demonstration of jml tools. Technical Report 05-13, Iowa State University, Department of Computer Science, 226 Atanasoff Hall, Ames IA 50011, April 2005. Available by anonymous ftp from ftp.cs.iastate.edu. [cited at p. 7, 8]

[12] Gary T. Leavens, K. Rustan M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs. Jml: notations and tools supporting detailed design in java. In *OOPSLA 2000 Companion, Minneapolis, Minnesota*, pages 105–106. ACM, October 2000. ftp://ftp.cs.iastate.edu/pub/techreports/TR00-15/TR.ps.gz. [cited at p. 2]

[13] Bertrand Meyer. Applying design by contract. *Computer*, 25(10):40–51, 1992. [cited at p. 2]

[14] Overture-Core-Team. Overture Web site. http://www.overturetool.org, 2007. [cited at p. 2]

[15] E. Poll and B.P.F. Jacobs. A logic for the java modeling language jml. In *Fundamental Approaches to Software Engineering : 4th International Conference, FASE 2001 : Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001, Genova, Italy, April 2-6*, page 284, Toernooiveld 1, 6525 ED Nijmegen, November 2000. Springer Berlin / Heidelberg. [cited at p. 2]

[16] Wikipedia. Wikipedia homepage. *http://en.wikipedia.org*. [cited at p. 2]

[17] Russel Winder and Graham Roberts. *Developing Java Software, 3rd Edition*. John Wiley & Sons, Inc., New York, NY, USA, 2006. [cited at p. 3, 12]

[18] Cui Ye. Improving jml's assignable clause analysis. Technical report, Iowa State University, August 2006. [cited at p. 3]

[19] Murali Sitaraman Yoonsik Cheon, Gary T. Leavens and Stephen Edwards. Model variables: Cleanly supporting abstraction in design by contract. Technical Report 03-10a, Department of Computer Science, Iowa State University, September 2003. In *Software – Practice & Experience*, 35(6):583–599, May 2005. Available from `archives.cs.iastate.edu`. [cited at p. 3]

Appendices

Appendix A

Specification of the JML Abstract Representation

In this appendix, it is possible to see the complete specification of the *JML* abstract syntax. The appendix is divided in several sections, each one containing a complete block of the specification. Furthermore, each type is commented in order to understand what it represents. This AST is then converted, using the *ASTGEN* (see 4.3) into *JAVA* and *VDM++* classes in order to be used within the specification.

A.1 *JML* Specifications

The abstract representation of *JML* specifications assumes the possibility of existing more than one file, which means more than one class, or even nested classes. Thus, a *JML* specifications can be represented as a sequence of *JML* classes, as represented below.

```
Specifications ::
        class_list : seq of WrappedJmlClass
        ;
```

Each *JML* class is composed by a package name, a refinement clause, the correspond type imports and by the class body itself.

```
WrappedJmlClass ::
        package : seq of char
        refine : seq of char
        imports_java : seq of Import
        imports_jml : seq of ModelImport
```

```
        class_val : Class
        ;
```

Each import is nothing more than a string. However, it is possible to have *JAVA* imports, for *JAVA* types, and *JML* imports, for the *JML* pure types.

```
Import ::
        import : seq of char
        ;

ModelImport ::
        model : bool
        import : seq of char
        ;
```

Concerning the class, a *JML* class can be represented by the *VDM-SL* type presented below. A class is composed by an access definition, which represents the class visibility, a keyword representing the class kind (interface, class, etc), the class name, both the interface and class inheritance clause and the sequence of definition blocks (operations, etc).

```
Class ::
  access : AccessDefinition
  kind : ClassKind
  identifier : Identifier
  inheritance_clause : [ClassInheritanceClause]
  interface_inheritance : [InterfaceInheritanceClause]
  class_body : seq of DefinitionBlock
  ;
```

A few types related to the class.

```
ClassKind =
        <CLASS> | <ABSTRACT> | <INTERFACE>
        ;

InterfaceInheritanceClause ::
        identifier_list : seq of Identifier
        ;

ClassInheritanceClause ::
        identifier_list : Identifier
        ;

AccessDefinition ::
        scope  : Scope
        ;
```

```
Scope =
  <PUBLIC> | <PRIVATE> | <PROTECTED> | <DEFAULT>
  ;
```

A definition block can be an instance variable block, a value block (static variables), an invariant block, an operation block or other definitions, which represents the constructs at the *VDM++* side that do not have an association at the *JML* side, and vice-versa.

```
DefinitionBlock =
        InstanceVariableDefinitions |
        ValueDefinitions |
        InvariantDefinitions |
        OperationDefinitions |
        OtherDefinitions
        ;

OtherDefinitions :: ;
```

A.2 Instance Variables

In this section, the instance variables block is presented. It is possible to have both *JML* and *JAVA* variables inside a specification.

```
InstanceVariableDefinitions ::
        jml_variables : seq of Variable
        java_variables : seq of Variable
        ;
```

Furthermore, each variable has a number of information fields, represented by the type below.

```
Variable ::
        access : AccessDefinition
        model : bool
        statickeyword : bool
        final : bool
        type : Type
        identifier : Identifier
        expression : [Expression]
        ;
```

A.3 Values

A value in *JML* a static variable, which corresponds to the *VDM++* value. Below, it is possible to see the abstract definition of values in *JML*.

A value definition block is a sequence of values.

```
ValueDefinitions ::
    value_list : seq of ValueDefinition
        ;
```

Each value has the following definition:

```
ValueDefinition ::
        access : AccessDefinition
        static_mod : bool
        final_mod : bool
        shape: ValueShape
        ;

ValueShape ::
        identifier : Identifier
        type : Type
        expression : Expression
        ;
```

A.4 Invariants

The following definition represents invariants in *JML*. It is possible to have a sequence of invariants, each one with a particular definition shown below.

```
InvariantDefinitions ::
        invariant_list : seq of InvariantDefinition;

InvariantDefinition ::
        access : AccessDefinition
        redundant : bool
        predicate : Expression
        ;
```

A.5 Operations

The operations block is composed by a number of operations, represented by a sequence.

```
OperationDefinitions ::
```

```
        operation_list : seq of OperationDefinition
        ;
```

Each operation is represented by a number of fields, already explained in chapter 4.

```
OperationDefinition ::
        trailer : [MethodSpecifications]
        access : AccessDefinition
        pure : bool
        statickeyword : bool
        final : bool
        returning_type : Type
        identifier : Identifier
        parameter_list : seq of Parameter
        body : [Body]
        ;
```

AN operation can have new specifications or an extension to other specifications (through inheritance). There are three kinds of specifications: normal, behaviour and exceptional specifications.

```
MethodSpecifications ::
        specs : Specs
        also  : Specs
        ;

Specs ::
        list : seq of OperationTrailer
        ;

OperationTrailer =
        BehaviourSpec |
        ExceptionalSpec |
        NormalSpec
        ;
```

Each kind of specifications has a scope and a number of trailers (for example, pre-conditions).

```
BehaviourSpec ::
        privacy : Scope
        list : seq of Trailers
        ;

ExceptionalSpec ::
        privacy : Scope
        list : seq of Trailers
```

```
        ;

NormalSpec ::
        privacy : Scope
        list : seq of Trailers
        ;
```

There are five possible trailers allowed by *JML*, listed below.

```
Trailers =
        EnsuresClause |
        AssignableClause |
        SignalsClause |
        SignalsOnlyClause |
        RequiresClause
        ;
```

Each of the five possible trailers are described below. The common field between them is an expression, which will be explained further on.

```
EnsuresClause ::
        ensures_expression : Expression
        ;

RequiresClause ::
        requires_expression : Expression
        ;

AssignableClause ::
        assignable_list : seq of Identifier
        ;

SignalsClause ::
        exception : Exception
        predicate : Expression
        ;

SignalsOnlyClause ::
        reference_types : seq of ReferenceType
        ;
```

Below, one can see some useful types used above, and besides that, the representation of the body (as string) because this connections does not consider the connection between implementation of operations.

```
ReferenceType =
        Identifier
        ;
```

```
Exception ::
        type : ExceptionType
        identifier : Identifier
        ;

Body ::
        body : seq of char
        ;

Parameter ::
        type : Type
        identifier : Identifier
        ;
```

A.6 Types

In this section, it is possible to see the representation of some of the types chosen to be used at the *JML* side. Below, one can see those types.

```
Type =
        CharType |
        BoolType |
        EnumerationType |
        IntegerType |
        FloatType |
        MapValueToValueType |
        SetValueType |
        SeqValueType |
        ObjectType |
        VoidType |
        TupleType |
        NatType |
        ClassType |
        ClassName
        ;
```

This first set of types represent *JML* classes, which will be mapped from *VDM++* composite types.

```
ClassName ::
        id : Name;

ClassType ::
        id : Identifier
        field_list : seq of Field;

Field ::
        id : Identifier
        type : Type;
```

This set of types represent *JML* types which will be mapped from *VDM++* simple types.

```
NatType ::
        limit : nat;

TupleType ::
        vals : seq of Type;

CharType :: ;

BoolType :: ;

IntegerType :: ;

EnumerationType ::
        enum_literal : EnumLiteral;

FloatType :: ;
```

Finally, *VDM++* complex types can be mapped to the *JML* types represented below.

```
MapValueToValueType ::
        dom_type : Type
        rng_type : Type;

SetValueType ::
        type : Type;

SeqValueType ::
        type : Type
        limit : nat;

ObjectType :: ;

ExceptionType ::
    type : Identifier;

VoidType :: ;
```

A.7 Expressions

In this section, a number of *JML* expressions are presented. This expressions are used to map from *VDM++* expressions, and are used in several parts of a specification, such as pre-conditions, invariants, etc.

```
Expression = BracketedExpression |
             IfExpression |
```

```
                        UnaryExpression |
                        BinaryExpression |
                        ForAllExpression |
                        ExistsExpression |
                        OldName |
                        NewExpression |
                        Name |
                        PostfixExpression |
                        SetEnumeration |
                        SequenceEnumeration |
                        MapEnumeration |
                        ApplyExpression |
                        FieldSelectExpression |
                        ArrayExpression |
                        ThrowExpression |
                        LiteralExpression |
                        BlockExpression |
                        ThisExpression |
                        InstanceOfExpression |
                        OldName
                        ;
```

The following types represent the *old* expression, the *instanceof* expression and the *this* expression.

```
OldName ::
        identifier : Identifier;

InstanceOfExpression ::
        type : Type
        expression : Expression;

ThisExpression :: ;
```

Below, one can find types representing the *JML* map enumeration, the sequence enumeration and the set enumeration.

```
MapEnumeration ::
        list : seq of MapLet;

MapLet ::
        dom_val : Expression
        rng_val : Expression;

SequenceEnumeration ::
        list : seq of Expression;

SetEnumeration ::
        list : seq of Expression;
```

The block expression represents the *JML* type from which the *VDM++* let expression maps to. It has bind information and an expression. Furthermore, one can see definitions for an array expression and a throw expression.

```
BlockExpression ::
        bind : seq of ValueShape
        return_expr : Expression;

ArrayExpression ::
        list : seq of Expression
        ;

ThrowExpression ::
        exception : Identifier
        params : seq of Parameter
        ;
```

Below, it is possible to see definitions for the apply expression, and the field select expression.

```
ApplyExpression ::
    expression : Expression
        expression_list : seq of Expression
        ;

FieldSelectExpression ::
    expression : Expression
        name : Name
        ;
```

The definitions present below represents the bracketed expression in *JML*, and the if-then-else clause.

```
BracketedExpression ::
        expression : Expression
        ;

IfExpression ::
        if_expression : Expression
        then_expression : Expression
        else_expression : Expression
        ;
```

The unary expression has an operator, defined further on, and an expression in which the operator will operate.

```
UnaryExpression ::
        operator : UnaryOperator
```

```
        expression : Expression
        ;
```

There are a number of unary operators allowed in *JML*. One can increment or decrement the value of some expression. Furthermore, it is possible to use the unary plus and minus operators. Besides that, it is possible to use the not, old, absolute value, floor, domain, range, inverse, cardinality, power, head, tail, length and elements. Each one of this operators will have to respect the type of the expression in which they are being applied.

```
UnaryOperator =
        <INCREMENT> | <DECREMENT> |
        <PLUS> | <MINUS> |
        <NOT> | <OLD> | <ABS> |
        <FLOOR> | <DOM> | <RNG> |
        <INVERSE> | <CARD> | <POWER> |
        <HD> | <TL> | <LEN> | <ELEMS>
        ;
```

The binary expression is similar to the unary expression, unless it has one more expression. The binary operators are explained further on.

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

It is possible to chose from a number of operators. They are represented by the type below.

```
BinaryOperator =
        <MINUS> | <PLUS> | <MULTIPLY> |
        <DIVIDE> | <REMAIN> | <LE> |
        <L> | <G> | <GE> | <EQ> | <NE> |
        <INSTANCEOF> | <LAND> | <LOR> |
        <IMPLY> | <IMPLYBACK> | <EQUIV> |
        <NOTEQUIV> | <MULEQ> | <DIVEQ> |
    <REMEQ> | <PLUSEQ> | <MINUSEQ> |
        <SUBTYPE> | <MUNION> | <DOMRESTTO> |
        <RNGRESTTO> | <COMP> | <INSET> |
        <NOTINSET> | <UNION> | <INTER> |
        <SUBSET> | <PROPERSUBSET> | <CONCAT>
        ;
```

Below, it is possible to see the definition of the quantification expressions *forall* and *exists*. They are similar in a way they both have quantifier declarations and

an expression.

```
ForAllExpression ::
        bind_list : QuantifierDeclaration
        expression : seq of Expression
        ;

ExistsExpression ::
        bind_list : QuantifierDeclaration
        expression : seq of Expression
        ;
```

Each quantifier declaration mentioned above can have a bound, a type and a sequence of variables.

```
QuantifierDeclaration ::
        bound : [BoundModifiers]
        type : Type
        vars : seq of Identifier
        ;

BoundModifiers =
        <NON_NULL> |
        <NULLABLE>;

OldName ::
        identifier : Identifier
        ;
```

From the definitions below, it is possible to see the definitions of the *new* expression and the name expression, which can represent a method name, for example.

```
NewExpression ::
        type : Type
        expression_list : seq of Expression
        ;

Name ::
        class_identifier : [Identifier]
        identifier : Identifier
        ;
```

Below, a postfix expression is represented.

```
PostfixExpression ::
        primary : seq of PrimaryExpressions
        operation : PostfixOperation
```

```
        ;

PostfixOperation =
        <INCREMENT> |
        <DECREMENT>
        ;
```

Furthermore, a *JML* primary expression is defined below.

```
PrimaryExpressions ::
        primary : PrimaryExpression
        suffix : [PrimarySuffix]
        ;
```

Some *JML* keywords are represented below.

```
SuperKeyword :: ;
ThisKeyword :: ;
NullKeyword :: ;
ResultKeyword :: ;
OldKeyword :: ;
NotAssignedKeyword :: ;
StarKeyword :: ;
ClassKeyword :: ;
```

The primary suffix in *JML* can be one of four different types, listed below.

```
PrimarySuffix =
        ThisKeyword |
        ClassKeyword |
        SuperKeyword |
        ExpressionsList
        ;
```

Moreover, one can see the definitions for the *JML* primary expressions and the literal expression, responsible for the usage of literals such as natural numbers, etc.

```
ExpressionsList ::
    list : seq of Expression
                ;

PrimaryExpression =
    PrimaryExpressionLiteral |
        PrimaryExpressionKeyword
        ;

PrimaryExpressionLiteral ::
```

```
    lit : Literal
        ;

LiteralExpression ::
        lit : Literal
        ;

PrimaryExpressionKeyword ::
    keyword : PrimaryExpressionOption
        ;
```

A primary expression can be one of the following types.

```
PrimaryExpressionOption =
        NameId |
        SuperKeyword |
        ThisKeyword |
        NullKeyword |
        ResultKeyword |
        OldKeyword |
        NotAssignedKeyword
        ;

NameId ::
    name : Identifier
        ;
```

A.8 Literals

In this section, the definition of *JML* literals can be found. As one can see below, there are a number of possible literals to use in *JML* specifications.

```
Literal =
        NumericalLiteral |
        FloatLiteral |
        EnumLiteral |
        BooleanLiteral |
        CharacterLiteral |
        StringLiteral |
        NullLiteral;
```

Below, one can see definitions about enumeration literals, numerical and float literals.

```
EnumLiteral     ::
        val : seq of char;
```

```
NumericalLiteral ::
        val : nat;

FloatLiteral ::
        val : real;
```

Furthermore, the definition of character and boolean literals is presented.

```
BooleanLiteral ::
        val : bool;

CharacterLiteral ::
        val : char;
```

Finally, it is presented the definition of string and null literals and the definition of an identifier, which is a sequence of characters (string).

```
StringLiteral ::
        val : seq of char;

NullLiteral :: ;

Identifier = seq of char
```

Appendix B

Specification of the VDM++ to JML Mapper

B.1 Mapper from *VDM++* to *JML*

Below, one can find the types and instance variables being used by the mapper.

```
class Vdm2Jml

types

public ClassId = seq of char;
public TypeInfo = map ClassId to JmlType;
public Information ::
  field_list : seq of JmlField
  invariant  : [JmlExpression];

instance variables

public hold_type_info : map ClassId to TypeInfo;

public interfaces_list : set of ClassId;

public classes_list : set of ClassId;

public to_class : map ClassId to Information;

operations

-----------------------------------------------
-- MULTIPLE INHERITANCE -----------------------
```

```
-----------------------------------------------

public init :
  OmlSpecifications ==>
  JmlSpecifications
init(specs) ==
  if preprocess(specs)
  then return eliminateMI(specs)
  else return build_jml(specs);


public eliminateMI :
  OmlSpecifications ==>
  JmlSpecifications
eliminateMI(specs) ==
  if canProceed(specs)
  then (eliminate(specs);
        build_jml(specs))
  else return new JmlSpecifications([]);


public canProceed :
  OmlSpecifications ==>
  bool
canProceed(specs) ==
  let cl = specs.getClassList(),
      s  = gatherInfo(cl)
  in result(s);


public result :
  seq of bool ==>
  bool
result(s) ==
  let c = countFalses(s)
  in if c > 1
     then return false
         else return true;


public countFalses :
  seq of bool ==>
  nat
countFalses(s) ==
  if len s = 0
  then return 0
  else if hd(s) = false
       then return 1 + countFalses(tl(s))
           else return countFalses(tl(s));
```

```
public gatherInfo :
  seq of OmlClass ==>
  seq of bool
gatherInfo(s) ==
  return [ gatherInfoHelper(s(i))
    | i in set inds s & hasMI(s(i))];


public gatherInfoHelper :
  OmlClass ==>
  bool
gatherInfoHelper(c) ==
  let body = c.getClassBody(),
      name = c.getIdentifier()
  in return gatherBlocksInfo(body,name);


public gatherBlocksInfo :
  seq of OmlDefinitionBlock *
  seq of char ==>
  bool
gatherBlocksInfo(s,n) ==
  let q = [ check(s(i),n) | i in set inds s]
  in return land(q);


public check :
  IOmlDefinitionBlock *
  seq of char ==>
  bool
check(b,n) ==
  cases true:
    (isofclass(OmlOperationDefinitions,b))
          -> return checkOperations(b,n),
        (isofclass(OmlInstanceVariableDefinitions,b))
          -> return checkInstanceVars(b),
        (isofclass(OmlFunctionDefinitions,b))
          -> return checkFunctions(b),
        others
          -> return true
  end;


public checkOperations :
  OmlOperationDefinitions *
  seq of char ==>
  bool
checkOperations(op,n) ==
  let l = op.getOperationList(),
      s = [ checkOperation(l(i),n) | i in set inds l]
  in return land(s);
```

```
public checkOperation :
  OmlOperationDefinition *
  seq of char ==>
  bool
checkOperation(op,n) ==
  let s = op.getShape()
  in cases true:
       (isofclass(OmlExplicitOperation,s))
             -> return checkExplicitOperation(s,n),
           (isofclass(OmlImplicitOperation,s))
             -> return checkImplicitOperation(s,n),
           (isofclass(OmlExtendedExplicitOperation,s))
             -> return checkExtendedExplicitOperation(s,n),
           others
             -> return true
         end;


public checkImplicitOperation :
  OmlImplicitOperation *
  seq of char ==>
  bool
checkImplicitOperation(op,n) ==
  let c = op.getIdentifier()
  in if(n = c)
     then return false
     else return true;


public checkExplicitOperation :
  OmlExplicitOperation *
  seq of char ==>
  bool
checkExplicitOperation(op,n) ==
  let name = op.getIdentifier(),
      body = op.getBody(),
          b = body.hasStatement()
  in return
      not(name = n) and
          not(b) and
          ( body.getNotYetSpecified() or
                body.getSubclassResponsibility());


public checkExtendedExplicitOperation :
  OmlExtendedExplicitOperation *
  seq of char ==>
  bool
checkExtendedExplicitOperation(op,n) ==
```

```
  let name = op.getIdentifier(),
      body = op.getBody(),
          b = body.hasStatement()
  in return
      not(name = n) and
          not(b) and
          ( body.getNotYetSpecified() or
                body.getSubclassResponsibility());


public checkInstanceVars :
  OmlInstanceVariableDefinitions ==>
  bool
checkInstanceVars(iv) ==
  let l = iv.getVariablesList()
  in if len l = 0
     then return true
         else return false;


public checkFunctions :
  OmlFunctionDefinitions ==>
  bool
checkFunctions(fu) ==
  let l = fu.getFunctionList(),
      s = [ checkFunction(l(i)) | i in set inds l]
  in land(s);


public checkFunction :
  OmlFunctionDefinition ==>
  bool
checkFunction(fd) ==
  let s = fd.getShape()
  in cases true:
       (isofclass(OmlExplicitFunction,s))
             -> return checkExplicitFunction(s),
       (isofclass(OmlImplicitFunction,s))
             -> return true,
       (isofclass(OmlExtendedExplicitFunction,s))
             -> return checkExtendedExplicitFunction(s),
       (isofclass(OmlTypelessExplicitFunction,s))
             -> return checkTypelessExplicitFunction(s),
          others
             -> return true
        end;


public checkExplicitFunction :
  OmlExplicitFunction ==>
  bool
```

```
checkExplicitFunction(f) ==
  let body = f.getBody()
  in if (body.hasFunctionBody() and
         (body.getNotYetSpecified() or
                  body.getSubclassResponsibility()))
     then return false
         else return true;


public checkExtendedExplicitFunction :
  OmlExtendedExplicitFunction ==>
  bool
checkExtendedExplicitFunction(f) ==
  let body = f.getBody()
  in if (body.hasFunctionBody() and
         (body.getNotYetSpecified() or
                  body.getSubclassResponsibility()))
     then return false
         else return true;


public checkTypelessExplicitFunction :
  OmlTypelessExplicitFunction ==>
  bool
checkTypelessExplicitFunction(f) ==
  let body = f.getBody()
  in if (body.hasFunctionBody() and
         (body.getNotYetSpecified() or
                  body.getSubclassResponsibility()))
     then return false
         else return true;


public eliminate :
  OmlSpecifications ==>
  ()
eliminate(specs) ==
  (interfaces_list := updateInterfaceList(specs);
   classes_list := updateClassList(specs))
  ;


public updateInterfaceList :
  OmlSpecifications ==>
  set of ClassId
updateInterfaceList(specs) ==
  let s = specs.getClassList(),
      l = [ let q = s(i).getInheritanceClause(),
                k = q.getIdentifierList(),
                          t = getInterfaces(k,specs)
                    in t
```

```
              | i in set inds s & hasMI(s(i))],
          q = conc l
  in return elems q;


public getInterfaces :
  seq of (seq of char) *
  OmlSpecifications ==>
  seq of (seq of char)
getInterfaces(s,specs) ==
  let c = specs.getClassList(),
      l = [ s(i) | i in set inds s
                 & isInterfaceIn(s(i),c)]
  in return l;


public isInterfaceIn :
  seq of char *
  seq of OmlClass ==>
  bool
isInterfaceIn(name,cls) ==
  let c = getClass(name,cls)
  in return gatherInfoHelper(c);


public getClass :
  seq of char *
  seq of OmlClass ==>
  OmlClass
getClass(n,s) ==
  if len s > 0
  then let head = hd(s) in
       if head.getIdentifier() = n
       then return hd(s)
       else getClass(n,tl(s))
  else return new OmlClass();


public updateClassList :
  OmlSpecifications ==>
  set of ClassId
updateClassList(specs) ==
  let s = specs.getClassList(),
      l = [ let q = s(i).getInheritanceClause(),
                    k = q.getIdentifierList(),
                                t = getClasses(k,specs)
                        in t
              | i in set inds s & hasMI(s(i))],
          q = conc l
  in return elems q;
```

```
public getClasses :
  seq of (seq of char) *
  OmlSpecifications ==>
  seq of (seq of char)
getClasses(s,specs) ==
  let c = specs.getClassList(),
      l = [ s(i) | i in set inds s
                & isClassIn(s(i),c)]
  in return l;


public isClassIn :
  seq of char *
  seq of OmlClass ==>
  bool
isClassIn(name,cls) ==
  let c = getClass(name,cls)
  in return not gatherInfoHelper(c);


------------------------------------------------
-- PRE-PROCESSOR (INHERITANCE) -----------------
------------------------------------------------


public preprocess :
  OmlSpecifications ==>
  bool
preprocess(specs) ==
  let l = specs.getClassList()
  in return hasMultipleInheritance(l);


public hasMultipleInheritance :
  seq of OmlClass ==>
  bool
hasMultipleInheritance(s) ==
  let q = [ hasMI(s(i)) | i in set inds s]
  in lor(q);


public lor :
  seq of bool ==>
  bool
lor(s) ==
  if len s = 0
  then return false
  else return hd(s) or lor(tl(s));
```

```
public hasMI :
  OmlClass ==>
  bool
hasMI(c) ==
  let ic = c.getInheritanceClause()
  in if c.hasInheritanceClause()
     then let l = len ic.getIdentifierList()
              in if l > 1
                        then return true
                        else return false
         else return false;


-----------------------------------------------
-- PRE-PROCESSOR (TYPES) ----------------------
-----------------------------------------------


public gatherTypeInfo :
  OmlSpecifications ==>
  bool
gatherTypeInfo(specs) ==
  let classes = specs.getClassList(),
      s = [ buildClassEntry(classes(i))
              | i in set inds classes ]
  in return land(s);


public land :
  seq of bool ==>
  bool
land(s) ==
  if len s = 0
  then return true
  else return hd(s) and land(tl(s));


public buildClassEntry :
  OmlClass ==>
  bool
buildClassEntry(c) ==
  let cn = c.getIdentifier(),
      bl = c.getClassBody(),
      ts = buildTypeInfo(bl),
          cl = toClassInformation(bl)
  in (
        hold_type_info := hold_type_info ++ { cn |-> ts };
                to_class := to_class ++ cl;
                return true;
     );
```

```
public toClassInformation :
  seq of IOmlDefinitionBlock ==>
  map ClassId to Information
toClassInformation(s) ==
  let l = [ s(i) | i in set inds s &
          isofclass(OmlTypeDefinitions,s(i))]
  in return buildToClassInfo(l);


public buildToClassInfo :
  seq of OmlTypeDefinitions ==>
  map ClassId to Information
buildToClassInfo(s) ==
  let l = [ s(i).getTypeList() | i in set inds s]
  in return buildToClass(conc l);


public buildToClass :
  seq of OmlTypeDefinition ==>
  map ClassId to Information
buildToClass(s) ==
  let l = [ s(i) | i in set inds s & selectShape(s(i).getShape())]
  in buildClassMap(l);


public buildClassMap :
  seq of OmlTypeDefinition ==>
  map ClassId to Information
buildClassMap(s) ==
  if len s = 0 then return {|->} else
  let fst = hd s,
      tail = tl s,
      shp = fst.getShape(),
          val = buildMapValue(shp),
          key = getKey(shp)
  in return { key |-> val} ++ buildClassMap(tail);


public buildMapValue :
  IOmlTypeShape ==>
  Information
buildMapValue(s) ==
  cases true:
    (isofclass(OmlComplexType,s))
          -> return getMapValue(s),
        others
          -> return mk_Information([],nil)
  end;
```

```
public getMapValue :
  OmlComplexType ==>
  Information
getMapValue(c) ==
  let fld = c.getFieldList(),
      iv = c.getInvariant(),
          field = convertFieldList(fld),
          ninv = convertInvariant(iv)
  in return mk_Information(field,ninv);


public convertFieldList :
  seq of OmlField ==>
  seq of JmlField
convertFieldList(s) ==
  return [ convertField(s(i)) | i in set inds s];


public convertField :
  OmlField ==>
  JmlField
convertField(f) ==
  let id = f.getIdentifier(),
      tp = f.getType(),
          ntp = convertType(tp),
          nid = convertId(id)
  in return new JmlField(nid,ntp);


public convertId :
  [seq of char] ==>
  seq of char
convertId(s) ==
  if s = nil then return [] else return s;


public convertInvariant :
  [OmlInvariant] ==>
  [JmlExpression]
convertInvariant(invar) ==
  if invar = nil then return nil else
  let iv = invar.getExpression(),
      ex = convertExpression(iv)
  in return ex;


public getKey :
  IOmlTypeShape ==>
  seq of char
getKey(s) ==
  cases true:
```

```
    (isofclass(OmlComplexType,s))
          -> return getKeyValue(s),
        others
          -> return []
  end;


public getKeyValue :
  OmlComplexType ==>
  seq of char
getKeyValue(c) ==
  return c.getIdentifier();


public selectShape :
  IOmlTypeShape ==>
  bool
selectShape(s) ==
  cases true:
    (isofclass(OmlComplexType,s))
          -> return true,
        others
          -> return false
  end;


public buildTypeInfo :
  seq of OmlDefinitionBlock ==>
  TypeInfo
buildTypeInfo(s) ==
  let q = collectTypeBlocks(s),
      r = collectTypeInfo(q)
  in return r;


public collectTypeBlocks :
  seq of IOmlDefinitionBlock ==>
  seq of IOmlTypeDefinitions
collectTypeBlocks(s) ==
  return [ s(i) | i in set inds s &
    isofclass(OmlTypeDefinitions,s(i))];


public collectTypeInfo :
  seq of OmlTypeDefinitions ==>
  TypeInfo
collectTypeInfo(s) ==
  let q = mergeTypeDefs(s),
      l = constructTypeInfo(q),
          m = mergeTypeInfos(l)
  in return m;
```

```
public mergeTypeDefs :
  seq of OmlTypeDefinitions ==>
  seq of OmlTypeDefinition
mergeTypeDefs(s) ==
  return conc [ s(i).getTypeList() | i in set inds s ];


public constructTypeInfo :
  seq of OmlTypeDefinition ==>
  seq of TypeInfo
constructTypeInfo(s) ==
  return [ buildTypeInfoElement(s(i)) | i in set inds s];


public buildTypeInfoElement :
  OmlTypeDefinition ==>
  TypeInfo
buildTypeInfoElement(e) ==
  let name = getTypeName(e),
      val = getConvertedType(e)
  in return {name |-> val};


public getTypeName :
  OmlTypeDefinition ==>
  seq of char
getTypeName(e) ==
  let s = e.getShape(),
      n = getNameFromShape(s)
  in return n;


public getNameFromShape :
  IOmlTypeShape ==>
  seq of char
getNameFromShape(s) ==
  cases true:
    (isofclass(OmlSimpleType,s))
          -> return getSimpleName(s),
        (isofclass(OmlComplexType,s))
          -> return getComplexName(s),
        others
          -> return []
  end;


public getSimpleName :
  OmlSimpleType ==>
  seq of char
```

```
getSimpleName(s) ==
  return s.getIdentifier();


public getComplexName :
  OmlComplexType ==>
  seq of char
getComplexName(s) ==
  return s.getIdentifier();


public getConvertedType :
  OmlTypeDefinition ==>
  JmlType
getConvertedType(e) ==
  let s = e.getShape()
  in cases true:
       (isofclass(OmlSimpleType,s))
             -> return getConvertedTypeHelper(s),
           others
             -> return getClassInformation(s)
     end;


public getConvertedTypeHelper :
  OmlSimpleType ==>
  JmlType
getConvertedTypeHelper(t) ==
  let tp = t.getType(),
      nt = convertType(tp)
  in return nt;


public getClassInformation :
  OmlComplexType ==>
  IJmlType
getClassInformation(t) ==
  let id = t.getIdentifier(),
      n = new JmlName(nil,id),
          c = new JmlClassName(n)
  in return c;


public mergeTypeInfos :
  seq of TypeInfo ==>
  TypeInfo
mergeTypeInfos(s) ==
  let r = elems s
  in return merge r;
```

```
-----------------------------------------------
-- MAPPER: VDM++ TO JML ------------------------
-----------------------------------------------

public build_jml :
  OmlSpecifications ==>
  JmlSpecifications
build_jml(specs) ==
  let classes = specs.getClassList(),
          jml_classes = convertVdm2JmlClasses(classes)
  in return new JmlSpecifications(jml_classes);


public convertVdm2JmlClasses :
  seq of OmlClass ==>
  seq of JmlWrappedJmlClass
convertVdm2JmlClasses(classes) ==
  return [ convertVdm2JmlClass(classes(i)) | i in set inds classes ];


public convertVdm2JmlClass :
  OmlClass ==>
  JmlWrappedJmlClass
convertVdm2JmlClass(c) ==
  let id   = c.getIdentifier(),
          inh  = c.getInheritanceClause(),
          body = c.getClassBody(),
          jmlclass = build_class(id, inh, body),
          jmlimports = getJmlImports(),
          javaimports = getJavaImports()
  in return new JmlWrappedJmlClass([],[],javaimports,jmlimports,jmlclass);


--FIXME: should search jml types, and add them
public getJmlImports :
  () ==>
  seq of JmlModelImport
getJmlImports() == return [];


--FIXME: should search java types, and add them
public getJavaImports :
  () ==>
  seq of JmlImport
getJavaImports() == return [];


public build_class :
  seq of char *
  OmlInheritanceClause *
  seq of OmlDefinitionBlock ==>
```

```
  JmlClass
build_class(id,inh,body) ==
  let scope = new JmlScope(3),
          access = new JmlAccessDefinition(scope),
          kind = new JmlClassKind(0),
          class_inh = getSuperClasses(inh),
          inter_inh = getInterfaces(inh),
          def_blocks = build_def_blocks(body)
  in return new JmlClass(access, kind, id, class_inh, inter_inh, def_blocks);


public getInterfaces :
  [OmlInheritanceClause] ==>
  [JmlInterfaceInheritanceClause]
getInterfaces(inh) ==
  if inh = nil then return nil else
  let list = inh.getIdentifierList()
  in if len list = 0
     then return nil
         else let lst = [ list(i) | i in set inds list
                        & list(i) in set interfaces_list]
                 in return new JmlInterfaceInheritanceClause(lst);


public getSuperClasses :
  [OmlInheritanceClause] ==>
  [JmlClassInheritanceClause]
getSuperClasses(inh) ==
  if inh = nil then return nil else
  let list = inh.getIdentifierList()
  in if len list = 0
     then return nil
         else let lst = [ list(i)
                        | i in set inds list
                                    & list(i) in set classes_list ]
                 in if len lst > 1 or len lst = 0
                       then return nil
                       else return new JmlClassInheritanceClause(hd lst);


public build_def_blocks :
  seq of IOmlDefinitionBlock ==>
  seq of IJmlDefinitionBlock
build_def_blocks(s) ==
  return [ build_def_block(s(i)) | i in set inds s];


-----------------------------------------------------
-- Values -----------------------------------------
-----------------------------------------------------
```

```
public build_def_block :
  OmlValueDefinitions ==>
  JmlValueDefinitions
build_def_block(t) ==
  let val = t.getValueList(),
          res = buildValueList(val)
  in return new JmlValueDefinitions(res);


public buildValueList :
  seq of OmlValueDefinition ==>
  seq of JmlValueDefinition
buildValueList(s) ==
  return [ buildJmlValue(s(i)) | i in set inds s];


public buildJmlValue :
  OmlValueDefinition ==>
  JmlValueDefinition
buildJmlValue(val) ==
  let access = val.getAccess(),
          shape = val.getShape(),
          newAccess = buildJmlAccess(access),
          newShape = buildJmlShape(shape)
  in return new JmlValueDefinition(newAccess,true,true,newShape);


public buildJmlAccess :
  OmlAccessDefinition ==>
  JmlAccessDefinition
buildJmlAccess(a) ==
  let scope = a.getScope(),
          res   = buildJmlScope(scope)
      in return new JmlAccessDefinition(res);


public buildJmlScope :
  OmlScope ==>
  JmlScope
buildJmlScope(s) ==
  return new JmlScope(s.getValue());


public buildJmlShape :
  OmlValueShape ==>
  JmlValueShape
buildJmlShape(s) ==
  let p = s.getPattern(),
          t = s.getType(),
          e = s.getExpression(),
```

```
          id = getIdentifier(p),
          tp = convertType(t),
          exp = convertExpression(e)
  in return new JmlValueShape(id, tp, exp);


public getIdentifier :
  OmlPatternIdentifier ==>
  seq of char
getIdentifier(p) ==
  return p.getIdentifier();


-----------------------------------------------------
-- Instance variables -------------------------------
-----------------------------------------------------


public build_def_block :
  OmlInstanceVariableDefinitions ==>
  JmlInstanceVariableDefinitions
build_def_block(v) ==
  let s = buildVariablesList(v)
  in return new JmlInstanceVariableDefinitions(s,[]);


public buildVariablesList :
  OmlInstanceVariableDefinitions ==>
  seq of [JmlVariable]
buildVariablesList(s) ==
  let q = s.getVariablesList()
  in return [ buildVariables(q(i)) |
     i in set inds q & is_(q(i),OmlInstanceVariable) ];


public buildVariables :
  OmlInstanceVariable ==>
  JmlVariable
buildVariables(var) ==
  let access = var.getAccess(),
      scope = access.getScope(),
      assign = var.getAssignmentDefinition(),
      newScope = new JmlScope(scope.getValue()),
      newAccess = new JmlAccessDefinition(newScope),
      stat = access.getStaticAccess(),
      tp = convertType(assign.getType()),
      id = assign.getIdentifier(),
      expr = convertExpression(assign.getExpression())
  in return new JmlVariable(newAccess, true, stat,
                            false, tp, id, expr);
```

```
-----------------------------------------------------
-- Operations -----------------------------------
-----------------------------------------------------


public build_def_block :
  OmlOperationDefinitions ==>
  JmlOperationDefinitions
build_def_block(t) ==
  let s = t.getOperationList(),
          res = convertOperationList(s)
  in return new JmlOperationDefinitions(res);


public convertOperationList :
  seq of OmlOperationDefinition ==>
  seq of JmlOperationDefinition
convertOperationList(s) ==
  return [ convertOperation(s(i)) | i in set inds s ];


public convertOperation :
  OmlOperationDefinition ==>
  JmlOperationDefinition
convertOperation(op) ==
  let access = op.getAccess(),
          scope = access.getScope(),
          static_val = access.getStaticAccess(),
          newScope = new JmlScope(scope.getValue()),
          newAccess = new JmlAccessDefinition(newScope),
          shape = op.getShape()
  in return buildJmlOperation(newAccess,static_val,shape);


public buildJmlOperation :
  JmlAccessDefinition *
  bool *
  IOmlOperationShape ==>
  JmlOperationDefinition
buildJmlOperation(access,statickey,shape) ==
  cases true:
    (isofclass(OmlImplicitOperation,shape))
          -> buildImplicitJmlOperation(access,statickey,shape),
        (isofclass(OmlExplicitOperation,shape))
          -> buildExplicitJmlOperation(access,statickey,shape),
        (isofclass(OmlExtendedExplicitOperation,shape))
          -> buildExtendedJmlOperation(access,statickey,shape)
  end;

public buildImplicitJmlOperation :
```

```
  JmlAccessDefinition *
  bool *
  OmlImplicitOperation ==>
  JmlOperationDefinition
buildImplicitJmlOperation(access,statickey,shape) ==
  let id = shape.getIdentifier(),
          params = shape.getPatternTypePairList(),
          returns = shape.getIdentifierTypePairList(),
          trailer = shape.getTrailer(),
          newTrailer = buildOperationTrailer(trailer,access),
          returnType = buildReturnType(returns),
          newParams = buildParameters(params)
  in return new JmlOperationDefinition(newTrailer,access,
       true,statickey,false,returnType,id,newParams,nil);


public buildExplicitJmlOperation :
  JmlAccessDefinition *
  bool *
  OmlExplicitOperation ==>
  JmlOperationDefinition
buildExplicitJmlOperation(access,statickey,shape) ==
  let id = shape.getIdentifier(),
      tp = shape.getType(),
      type_rng = getRngType(tp),
      params = shape.getParameterList(),
      trailer = buildOperationTrailer(shape.getTrailer(),access),
      paramsList = buildOperationParameters(params,tp)
  in return new JmlOperationDefinition(trailer,access,true,
     statickey,false,type_rng,id,paramsList,nil);


public buildExtendedJmlOperation :
  JmlAccessDefinition *
  bool *
  OmlExtendedExplicitOperation ==>
  JmlOperationDefinition
buildExtendedJmlOperation(access,statickey,shape) ==
  let id = shape.getIdentifier(),
      params = shape.getPatternTypePairList(),
          returns = shape.getIdentifierTypePairList(),
          trailer = shape.getTrailer(),
          newTrailer = buildOperationTrailer(trailer,access),
          returnType = buildReturnType(returns),
          newParams = buildParameters(params)
  in return new JmlOperationDefinition(newTrailer,access,true,
     statickey,false,returnType,id,newParams,nil);


public buildOperationParameters :
  seq of OmlPattern *
```

```
  OmlOperationType ==>
  seq of JmlParameter
buildOperationParameters(s,t) ==
  let domtype = t.getDomType()
  in return buildOperationParameter(s,domtype);


public buildOperationParameter :
  seq of OmlPattern *
  IOmlType ==>
  seq of JmlParameter
buildOperationParameter(s,t) ==
  cases true:
    (isofclass(OmlProductType,t))
          -> return unfoldProductType(s,t),
        others
          -> return buildJmlParameter(s,t)
  end;


public unfoldProductType :
  seq of OmlPattern *
  OmlProductType ==>
  seq of JmlParameter
unfoldProductType(s,t) ==
  let ts = buildSeqTypes(t),
      p  = extractPatterns(s),
          f  = buildParameterSeq(p,ts)
  in return f;


public buildSeqTypes :
  OmlProductType ==>
  seq of OmlType
buildSeqTypes(p) ==
  let rhs = p.getRhsType(),
      lhs = p.getLhsType()
  in return (extractProductType(rhs) ^
  extractProductType(lhs));


public extractProductType :
  IOmlType ==>
  seq of OmlType
extractProductType(t) ==
  cases true:
    (isofclass(OmlProductType,t))
          -> return buildSeqTypes(t),
        others
          -> return [ t ]
  end;
```

```
public extractPatterns :
  seq of OmlPattern ==>
  seq of (seq of char)
extractPatterns(s) ==
  return [ extractPattern(s(i)) | i in set inds s];


public extractPattern :
  IOmlPattern ==>
  seq of char
extractPattern(p) ==
  cases true:
    (isofclass(OmlPatternIdentifier,p))
          -> return getId(p),
        others
          -> return []
  end;


public buildParameterSeq :
  seq of (seq of char) *
    seq of OmlType ==>
  seq of JmlParameter
buildParameterSeq(p,t) ==
  if len p <> len t
  then return []
  else return
    [ let nt = convertType(t(i)),
              v = p(i)
          in new JmlParameter(nt,v)
        | i in set inds p];


public buildJmlParameter :
  seq of OmlPattern *
  OmlType ==>
  seq of JmlParameter
buildJmlParameter(s,t) ==
  if len s = 0 or len s > 1
  then return []
  else
  let tp = convertType(t),
      fst = hd(s),
          id = getIds(fst),
          par = new JmlParameter(tp,id)
  in return [par];


public getIds :
```

```
  IOmlPattern ==>
  seq of char
getIds(p) ==
  cases true:
    (isofclass(OmlPatternIdentifier,p))
          -> return getId(p),
        others
          -> return []
  end;


public getId :
  OmlPatternIdentifier ==>
  seq of char
getId(p) ==
  return p.getIdentifier();


public buildReturnType :
  seq of OmlIdentifierTypePair ==>
  IJmlType
buildReturnType(s) ==
  if len s > 1
  then let q = [ convertType(s(i).getType()) | i in set inds s ],
           t = new JmlTupleType(q)
           in return t
  else let t = s(1).getType()
       in return convertType(t);


public buildParameters :
  seq of OmlPatternTypePair ==>
  seq of JmlParameter
buildParameters(s) ==
  return conc [ buildParameter(s(i)) | i in set inds s];


public buildParameter :
  OmlPatternTypePair ==>
  seq of JmlParameter
buildParameter(p) ==
  let s = p.getPatternList(),
          t = p.getType()
  in return [ buildParam(s(i),t) | i in set inds s];


public buildParam :
  OmlPatternIdentifier *
  OmlType ==>
  JmlParameter
buildParam(p,t) ==
```

```
  let tp = convertType(t),
          id = p.getIdentifier()
  in return new JmlParameter(tp,id);


public buildOperationTrailer :
  OmlOperationTrailer *
  JmlAccessDefinition ==>
  [JmlMethodSpecifications]
buildOperationTrailer(trailer,access) ==
  let ex = trailer.getExternals(),
          pr = trailer.getPreExpression(),
          po = trailer.getPostExpression(),
          ep = trailer.getExceptions()
  in let normal = buildNormalBehaviour(ex,pr,po,ep,
                   access.getScope().getValue()),
                  spec = new JmlSpecs(normal),
                  also = new JmlSpecs([])
         in return new JmlMethodSpecifications(spec,also);


public buildError :
  OmlError ==>
  JmlRequiresClause
buildError(e) ==
  let expr = convertExpression(e.getLhs())
  in return new JmlRequiresClause(expr);


public buildNormalBehaviour :
  [OmlExternals] *
  [OmlExpression] *
  [OmlExpression] *
  [OmlExceptions] *
  nat ==>
  seq of JmlNormalSpec
buildNormalBehaviour(ex,pr,po,ep,n) ==
  let externals = buildAssignableClause(ex),
      pres = buildRequiresClause(pr,ep),
          posts = buildEnsuresClause(po,ep),
          trailers = externals ^ pres ^ posts,
          scope = new JmlScope(n)
  in return [new JmlNormalSpec(scope,trailers) ];


--------------------------
-- Assignable Clause -----
--------------------------


public buildAssignableClause :
```

```
  [OmlExternals] ==>
  seq of JmlAssignableClause
buildAssignableClause(e) ==
  if e = nil then return [] else
  let list = e.getExtList(),
      res = [ getExternalNames(list(i).getNameList())
                  | i in set inds list ]
  in return res;


public getExternalNames :
  seq of OmlName ==>
  JmlAssignableClause
getExternalNames(s) ==
  let l = [ s(i).getIdentifier() | i in set inds s]
  in return new JmlAssignableClause(l);


--------------------------
-- Requires Clause -------
--------------------------


public buildRequiresClause :
  [OmlExpression] *
  [OmlExceptions] ==>
  seq of JmlRequiresClause
buildRequiresClause(pres,exc) ==
  if (pres = nil and exc = nil) then return [] else
  let exceptions = buildDisjunctions(exc),
      precond = convertExpression(pres),
          final_pre = chooserDisj(exceptions,precond),
          req = new JmlRequiresClause(final_pre)
  in return [ req ];


public chooserDisj :
  [IJmlExpression] *
  [IJmlExpression] ==>
  IJmlExpression
chooserDisj(exc,expr) ==
  cases true:
    (exc = nil) -> return expr,
        (expr = nil) -> return exc,
        others ->
          let op = new JmlBinaryOperator(1),
          final_pre = new JmlBinaryExpression(expr,op,exc)
        in return final_pre
  end;
```

```
public buildDisjunctions :
  [OmlExceptions] ==>
  [IJmlExpression]
buildDisjunctions(exc) ==
  if exc = nil then return nil else
  let l = exc.getErrorList()
  in if len l = 0
     then let lit = new JmlBooleanLiteral(true),
                  expr = new JmlLiteralExpression(lit)
                  in return expr
     else return buildDisjunction(l);


public buildDisjunction :
  seq of OmlError ==>
  IJmlExpression
buildDisjunction(s) ==
  if len s = 0
  then let lit = new JmlBooleanLiteral(true),
           exp = new JmlLiteralExpression(lit)
           in return exp
  else let erro = hd(s),
           e = erro.getLhs(),
                   expr = convertExpression(e),
                   op = new JmlBinaryOperator(1),
                   rhs = buildDisjunction(tl(s))
           in return new JmlBinaryExpression(expr,op, rhs);


--------------------------
-- Ensures Clause --------
--------------------------


public buildEnsuresClause :
  [OmlExpression] *
  [OmlExceptions] ==>
  seq of JmlEnsuresClause
buildEnsuresClause(po,ex) ==
  if (po = nil and ex = nil) then return [] else
  let postexpr = convertExpression(po),
      conj = buildConjunctions(ex),
          finalpost = chooserConj(conj,postexpr),
          ens = new JmlEnsuresClause(finalpost)
  in return [ ens ];


public chooserConj :
  [IJmlExpression] *
  [IJmlExpression] ==>
  IJmlExpression
```

```
chooserConj(exc,expr) ==
  cases true:
    (exc = nil) -> return expr,
        (expr = nil) -> return exc,
        others ->
          let op = new JmlBinaryOperator(11),
              finalpost = new JmlBinaryExpression(expr,op,exc)
        in return finalpost
  end;


public buildConjunctions :
  [OmlExceptions] ==>
  [IJmlExpression]
buildConjunctions(except) ==
  if except = nil then return nil else
  let errlst = except.getErrorList()
  in if len errlst = 0
     then let lit = new JmlBooleanLiteral(true),
                  expr = new JmlLiteralExpression(lit)
                  in return expr
     else return buildConjunction(errlst);


public buildConjunction :
  seq of OmlError ==>
  IJmlExpression
buildConjunction(s) ==
  if len s = 0
  then let lit = new JmlBooleanLiteral(true),
           exp = new JmlLiteralExpression(lit)
           in return exp
  else let erro = hd(s),
           e = erro.getLhs(),
                  expr = convertExpression(e),
                  op = new JmlBinaryOperator(11),
                  rhs = buildConjunction(tl(s))
           in return new JmlBinaryExpression(expr,op, rhs);

-----------------------------------------------------
-- Other constructs --------------------------------
-----------------------------------------------------


--FIXME: a function can be mapped. Think about this.
public build_def_block :
  OmlFunctionDefinitions ==>
  JmlOtherDefinitions
build_def_block(-) == return new JmlOtherDefinitions();
```

```
public build_def_block :
  OmlSynchronizationDefinitions ==>
  JmlOtherDefinitions
build_def_block(-) == return new JmlOtherDefinitions();


public build_def_block :
  OmlThreadDefinition ==>
  JmlOtherDefinitions
build_def_block(-) == return new JmlOtherDefinitions();


public build_def_block :
  OmlTraceDefinitions ==>
  JmlOtherDefinitions
build_def_block(-) == return new JmlOtherDefinitions();


--ignored here, but pre-processed earlier.
public build_def_block :
  OmlTypeDefinitions ==>
  JmlOtherDefinitions
build_def_block(-) == return new JmlOtherDefinitions();


-----------------------------------------------------
-- Convert Types ------------------------------------
-----------------------------------------------------


public convertType :
  IOmlType ==>
  IJmlType
convertType(t) ==
  cases true:
    (isofclass(OmlBracketedType,t))
          -> return convertBracketedType(t),
        (isofclass(OmlBoolType,t))
          -> return new JmlBoolType(),
        (isofclass(OmlNatType,t))
          -> return new JmlNatType(0),
        (isofclass(OmlNat1Type,t))
          -> return new JmlNatType(1),
        (isofclass(OmlIntType,t))
          -> return new JmlIntegerType(),
        (isofclass(OmlRealType,t))
          -> return new JmlFloatType(),
        (isofclass(OmlCharType,t))
          -> return new JmlCharType(),
        (isofclass(OmlTokenType,t))
          -> return new JmlObjectType(),
```

```
        (isofclass(OmlQuoteType,t))
          -> return convertQuoteType(t),
        (isofclass(OmlCompositeType,t))
          -> return convertCompositeType(t),
        (isofclass(OmlProductType,t))
          -> return convertProductType(t),
        (isofclass(OmlSetType,t))
          -> return convertSetType(t),
        (isofclass(OmlSeq0Type,t))
          -> return convertSeq0Type(t),
        (isofclass(OmlSeq1Type,t))
          -> return convertSeq1Type(t),
        (isofclass(OmlGeneralMapType,t))
          -> return convertMapType(t),
        (isofclass(OmlEmptyType,t))
          -> return new JmlVoidType(),
        (isofclass(OmlTypeName,t))
          -> convertTypeName(t),
        others
          -> return new JmlVoidType()
  end;


public convertBracketedType :
  OmlBracketedType ==>
  JmlType
convertBracketedType(t) ==
  return convertType(t.getType());


public convertQuoteType :
  OmlQuoteType ==>
  JmlEnumerationType
convertQuoteType(t) ==
  let q = t.getQuoteLiteral(),
      id = q.getVal(),
      l = new JmlEnumLiteral(id)
  in return new JmlEnumerationType(l);


public convertProductType :
  OmlProductType ==>
  JmlTupleType
convertProductType(t) ==
  let tp = buildSeqTypes(t),
      sq = [ convertType(tp(i)) | i in set inds tp ]
  in return new JmlTupleType(sq);


public convertCompositeType :
  OmlCompositeType ==>
```

```
  JmlClassType
convertCompositeType(t) ==
  let id = t.getIdentifier()
  in return new JmlClassType(id,[]);


public convertSetType :
  OmlSetType ==>
  JmlSetValueType
convertSetType(t) ==
  let tp = t.getType(),
      newtp = convertType(tp)
  in return new JmlSetValueType(newtp);


public convertSeq0Type :
  OmlSeq0Type ==>
  JmlSeqValueType
convertSeq0Type(t) ==
  let tp = t.getType(),
      newtp = convertType(tp)
  in return new JmlSeqValueType(newtp,0);


public convertSeq1Type :
  OmlSeq0Type ==>
  JmlSeqValueType
convertSeq1Type(t) ==
  let tp = t.getType(),
      newtp = convertType(tp)
  in return new JmlSeqValueType(newtp,1);


public convertMapType :
  OmlGeneralMapType ==>
  JmlMapValueToValueType
convertMapType(t) ==
  let mapdom = t.getDomType(),
          maprng = t.getRngType(),
      newdom = convertType(mapdom),
      newrng = convertType(maprng)
  in return new JmlMapValueToValueType(newdom,newrng);


public convertTypeName :
  OmlTypeName ==>
  JmlClassName
convertTypeName(t) ==
  let id = t.getName(),
      newid = convertName(id)
  in return new JmlClassName(newid);
```

```
public getDomType :
  OmlOperationType ==>
  JmlType
getDomType(op) ==
  return convertType(op.getDomType());


public getRngType :
  OmlOperationType ==>
  JmlType
getRngType(op) ==
  return convertType(op.getRngType());


-----------------------------------------------------
-- Convert Expressions ------------------------------
-----------------------------------------------------


public convertExpression :
  [IOmlExpression] ==>
  [IJmlExpression]
convertExpression(e) ==
  if e = nil then return nil else
  cases true:
    (isofclass(OmlBracketedExpression,e))
          -> return convertBracketedExpression(e),
    (isofclass(OmlLetExpression,e))
          -> return convertLetExpression(e),
        (isofclass(OmlIfExpression,e))
          -> return convertIfExpression(e),
        (isofclass(OmlUnaryExpression,e))
          -> return convertUnaryExpression(e),
        (isofclass(OmlBinaryExpression,e))
          -> return convertBinaryExpression(e),
        (isofclass(OmlForAllExpression,e))
          -> return convertForAllExpression(e),
        (isofclass(OmlExistsExpression,e))
          -> return convertExistsExpression(e),
        (isofclass(OmlSetEnumeration,e))
          -> return convertSetEnumeration(e),
        (isofclass(OmlSequenceEnumeration,e))
          -> return convertSequenceEnumeration(e),
        (isofclass(OmlMapEnumeration,e))
          -> return convertMapEnumeration(e),
        (isofclass(OmlTupleConstructor,e))
          -> return convertTupleConstructor(e),
        (isofclass(OmlRecordConstructor,e))
          -> return convertRecordExpression(e),
```

```
        (isofclass(OmlApplyExpression,e))
          -> return convertApplyExpression(e),
        (isofclass(OmlFieldSelect,e))
          -> return convertFieldSelect(e),
        (isofclass(OmlNewExpression,e))
          -> return convertNewExpression(e),
        (isofclass(OmlSelfExpression,e))
          -> return convertSelfExpression(e),
        (isofclass(OmlIsExpression,e))
          -> return convertIsExpression(e),
        (isofclass(OmlUndefinedExpression,e))
          -> return convertUndefinedExpression(e),
        (isofclass(OmlIsofclassExpression,e))
          -> return convertIsOfClassExpression(e),
        (isofclass(OmlName,e))
          -> return convertName(e),
        (isofclass(OmlOldName,e))
          -> return convertOldName(e),
        (isofclass(OmlSymbolicLiteralExpression,e))
          -> return convertLiteralExpression(e)
  end;


public convertBracketedExpression :
  OmlBracketedExpression ==>
  JmlBracketedExpression
convertBracketedExpression(e) ==
  let exp = e.getExpression(),
      newexpr = convertExpression(exp)
  in return new JmlBracketedExpression(newexpr);


public convertLetExpression :
  OmlLetExpression ==>
  JmlBlockExpression
convertLetExpression(e) ==
  let bind = e.getDefinitionList(),
      newbind = buildJmlShapes(bind),
          expr = e.getExpression(),
          res = convertExpression(expr)
  in return new JmlBlockExpression(newbind,res);


public buildJmlShapes :
  seq of OmlValueShape ==>
  seq of JmlValueShape
buildJmlShapes(s) ==
  return [ buildJmlShape(s(i)) | i in set inds s];


public convertIfExpression :
```

```
  OmlIfExpression ==>
  JmlIfExpression
convertIfExpression(e) ==
  let if_expr = e.getIfExpression(),
      then_expr = e.getThenExpression(),
          else_expr = e.getElseExpression(),
          newif = convertExpression(if_expr),
          newthen = convertExpression(then_expr),
          newelse = convertExpression(else_expr)
  in return new JmlIfExpression(newif,newthen,newelse);


public convertUnaryExpression :
  OmlUnaryExpression ==>
  JmlUnaryExpression
convertUnaryExpression(e) ==
  let op = e.getOperator(),
      newop = convertUnaryOperator(op),
          expr = e.getExpression(),
          newexpr = convertExpression(expr)
  in return new JmlUnaryExpression(newop,newexpr);


public convertUnaryOperator :
  OmlUnaryOperator ==>
  JmlUnaryOperator
convertUnaryOperator(op) ==
  let val = op.getValue()
  in cases true:
       (val = 0) -> return new JmlUnaryOperator(4),
           (val = 1) -> return new JmlUnaryOperator(5),
           (val = 2) -> return new JmlUnaryOperator(10),
           (val = 3) -> return new JmlUnaryOperator(7),
           (val = 5) -> return new JmlUnaryOperator(2),
           (val = 6) -> return new JmlUnaryOperator(0),
           (val = 8) -> return new JmlUnaryOperator(6),
           (val = 9) -> return new JmlUnaryOperator(14),
           (val = 10) -> return new JmlUnaryOperator(12),
           (val = 11) -> return new JmlUnaryOperator(8),
           (val = 14) -> return new JmlUnaryOperator(15),
           (val = 15) -> return new JmlUnaryOperator(1),
           (val = 16) -> return new JmlUnaryOperator(13),
           (val = 17) -> return new JmlUnaryOperator(3),
           others -> return new JmlUnaryOperator()
     end ;


public convertBinaryExpression :
  OmlBinaryExpression ==>
  JmlBinaryExpression
convertBinaryExpression(e) ==
```

```
  let lhs = e.getLhsExpression(),
      op = e.getOperator(),
          rhs = e.getRhsExpression(),
          newlhs = convertExpression(lhs),
          newrhs = convertExpression(rhs),
          newop = convertBinaryOperator(op)
  in return new JmlBinaryExpression(newlhs,newop,newrhs);


public convertBinaryOperator :
  OmlBinaryOperator ==>
  JmlBinaryOperator
convertBinaryOperator(op) ==
  let val = op.getValue()
  in cases true:
       (val = 1) -> return new JmlBinaryOperator(18),
           (val = 2) -> return new JmlBinaryOperator(23),
           (val = 3) -> return new JmlBinaryOperator(13),
           (val = 6) -> return new JmlBinaryOperator(22),
           (val = 7) -> return new JmlBinaryOperator(33),
           (val = 8) -> return new JmlBinaryOperator(1),
           (val = 10) -> return new JmlBinaryOperator(16),
           (val = 11) -> return new JmlBinaryOperator(0),
           (val = 12) -> return new JmlBinaryOperator(12),
           (val = 13) -> return new JmlBinaryOperator(4),
           (val = 14) -> return new JmlBinaryOperator(15),
           (val = 16) -> return new JmlBinaryOperator(28),
           (val = 19) -> return new JmlBinaryOperator(6),
           (val = 20) -> return new JmlBinaryOperator(34),
           (val = 21) -> return new JmlBinaryOperator(32),
           (val = 22) -> return new JmlBinaryOperator(2),
           (val = 23) -> return new JmlBinaryOperator(8),
           (val = 24) -> return new JmlBinaryOperator(30),
           (val = 25) -> return new JmlBinaryOperator(31),
           (val = 26) -> return new JmlBinaryOperator(14),
           (val = 27) -> return new JmlBinaryOperator(7),
           (val = 30) -> return new JmlBinaryOperator(21),
           (val = 31) -> return new JmlBinaryOperator(24),
           (val = 32) -> return new JmlBinaryOperator(19),
           (val = 33) -> return new JmlBinaryOperator(9),
           others -> return new JmlBinaryOperator()
     end;


public convertForAllExpression :
  OmlForAllExpression ==>
  JmlForAllExpression
convertForAllExpression(e) ==
  let bind = e.getBindList(),
      expr = e.getExpression(),
          newexpr = convertExpression(expr)
```

```
  in return buildForAllExpression(bind,newexpr);


public buildForAllExpression :
  seq of OmlBind *
  JmlExpression ==>
  JmlForAllExpression
buildForAllExpression(bind,expr) ==
  if len bind > 1
  then let b = buildBind(hd bind),
               e = buildForAllExpression(tl bind,expr)
           in return new JmlForAllExpression(b,[e])
  else let b = buildBind(hd bind)
       in return new JmlForAllExpression(b,[expr]);


public buildBind :
  IOmlBind ==>
  JmlQuantifierDeclaration
buildBind(b) ==
  cases true:
    (isofclass(OmlTypeBind,b))
          -> return buildTypeBinds(b),
        others
          -> return new JmlQuantifierDeclaration()
  end;


public buildTypeBinds :
  OmlTypeBind ==>
  JmlQuantifierDeclaration
buildTypeBinds(b) ==
  let p = b.getPattern(),
      t = b.getType()
  in buildTypeBind(p,t);


public buildTypeBind :
  seq of IOmlPattern *
  IOmlType ==>
  JmlQuantifierDeclaration
buildTypeBind(p,t) ==
  let s = getVars(p),
      t1 = convertType(t)
  in return new JmlQuantifierDeclaration(nil,t1,s);


public getVars :
  seq of IOmlPattern ==>
  seq of (seq of char)
getVars(s) ==
```

```
  return [ getVar(s(i)) | i in set inds s];


public getVar :
  IOmlPattern ==>
  seq of char
getVar(p) ==
  cases true:
    (isofclass(OmlPatternIdentifier,p))
          -> return getId(p),
        others
          -> return []
  end;


public convertExistsExpression :
  OmlExistsExpression ==>
  JmlExistsExpression
convertExistsExpression(e) ==
  let bind = e.getBindList(),
      expr = e.getExpression(),
          newexpr = convertExpression(expr)
  in return buildExistsExpression(bind,newexpr);


public buildExistsExpression :
  seq of OmlBind *
  JmlExpression ==>
  JmlExistsExpression
buildExistsExpression(bind,expr) ==
  if len bind > 1
  then let b = buildBind(hd bind),
               e = buildExistsExpression(tl bind,expr)
           in return new JmlExistsExpression(b,[e])
  else let b = buildBind(hd bind)
       in return new JmlExistsExpression(b,[expr]);


public convertSetEnumeration :
  OmlSetEnumeration ==>
  JmlSetEnumeration
convertSetEnumeration(e) ==
  let v = e.getExpressionList(),
      s = [ convertExpression(v(i)) | i in set inds v]
  in return new JmlSetEnumeration(s);


public convertSequenceEnumeration :
  OmlSequenceEnumeration ==>
  JmlSequenceEnumeration
convertSequenceEnumeration(e) ==
```

```
  let v = e.getExpressionList(),
      s = [ convertExpression(v(i)) | i in set inds v]
  in return new JmlSequenceEnumeration(s);


public convertMapEnumeration :
  OmlMapEnumeration ==>
  JmlMapEnumeration
convertMapEnumeration(e) ==
  let m = e.getMapletList(),
      s = convertMapLetList(m)
  in return new JmlMapEnumeration(s);


public convertMapLetList :
  seq of OmlMaplet ==>
  seq of JmlMapLet
convertMapLetList(s) ==
  return [ convertMapLet(s(i)) | i in set inds s];


public convertMapLet :
  OmlMaplet ==>
  JmlMapLet
convertMapLet(e) ==
  let d = e.getDomExpression(),
      r = e.getRngExpression(),
          dm = convertExpression(d),
          rn = convertExpression(r)
  in return new JmlMapLet(dm,rn);


public convertTupleConstructor :
  OmlTupleConstructor ==>
  JmlNewExpression
convertTupleConstructor(e) ==
  let s = e.getExpressionList(),
      q = convertExpressionList(s),
          t = new JmlTupleType()
  in return new JmlNewExpression(t,q);


public convertExpressionList :
  seq of OmlExpression ==>
  seq of JmlExpression
convertExpressionList(s) ==
  return [ convertExpression(s(i)) | i in set inds s];


public convertRecordExpression :
  OmlRecordConstructor ==>
```

```
  JmlNewExpression
convertRecordExpression(e) ==
  let n = e.getName(),
      s = e.getExpressionList(),
          q = convertExpressionList(s),
          nn = convertName(n),
          t = new JmlClassName(nn)
  in return new JmlNewExpression(t,q);


public convertApplyExpression :
  OmlApplyExpression ==>
  JmlApplyExpression
convertApplyExpression(e) ==
  let exp = e.getExpression(),
      lst = e.getExpressionList(),
          nexp = convertExpression(exp),
          nlst = convertExpressionList(lst)
  in return new JmlApplyExpression(nexp,nlst);


public convertFieldSelect :
  OmlFieldSelect ==>
  JmlFieldSelectExpression
convertFieldSelect(e) ==
  let exp = e.getExpression(),
      n = e.getName(),
      nexp = convertExpression(exp),
          nn = convertName(n)
  in return new JmlFieldSelectExpression(nexp,nn);


public convertName :
  OmlName ==>
  JmlName
convertName(n) ==
  let c = n.getClassIdentifier(),
      id = n.getIdentifier()
  in return new JmlName(c,id);


public convertNewExpression :
  OmlNewExpression ==>
  JmlNewExpression
convertNewExpression(e) ==
  let n = e.getName(),
      l = e.getExpressionList(),
          nn = convertName(n),
          t = new JmlClassName(nn),
          nl = convertExpressionList(l)
  in return new JmlNewExpression(t,nl);
```

```
public convertSelfExpression :
  OmlSelfExpression ==>
  JmlThisExpression
convertSelfExpression(-) ==
  return new JmlThisExpression();


public convertIsExpression :
  OmlIsExpression ==>
  JmlInstanceOfExpression
convertIsExpression(e) ==
  let t = e.getType(),
      exp = e.getExpression(),
          nt = convertType(t),
          nexp = convertExpression(exp)
  in return new JmlInstanceOfExpression(nt,nexp);


public convertUndefinedExpression :
  OmlUndefinedExpression ==>
  JmlUndefinedExpression
convertUndefinedExpression(-) ==
  return new JmlUndefinedExpression();


public convertIsOfClassExpression :
  OmlIsofclassExpression ==>
  JmlInstanceOfExpression
convertIsOfClassExpression(e) ==
  let n = e.getName(),
      nn = convertName(n),
          t = new JmlClassName(nn),
          exp = e.getExpression(),
          nexp = convertExpression(exp)
  in return new JmlInstanceOfExpression(t,nexp);


public convertOldName :
  OmlOldName ==>
  JmlOldName
convertOldName(n) ==
  return new JmlOldName(n.getIdentifier());


public convertLiteralExpression :
  OmlSymbolicLiteralExpression ==>
  JmlLiteralExpression
convertLiteralExpression(e) ==
  let lit = e.getLiteral(),
```

```
      nlit = convertLiteral(lit)
  in return new JmlLiteralExpression(nlit);


-----------------------------------------------------
-- Convert Literals ---------------------------------
-----------------------------------------------------


public convertLiteral :
  IOmlLiteral ==>
  IJmlLiteral
convertLiteral(l) ==
  cases true:
    (isofclass(OmlNumericLiteral,l))
          -> convertNumericLiteral(l),
        (isofclass(OmlRealLiteral,l))
          -> convertRealLiteral(l),
        (isofclass(OmlBooleanLiteral,l))
          -> convertBooleanLiteral(l),
        (isofclass(OmlNilLiteral,l))
          -> convertNilLiteral(l),
        (isofclass(OmlCharacterLiteral,l))
          -> convertCharacterLiteral(l),
        (isofclass(OmlTextLiteral,l))
          -> convertTextLiteral(l),
        (isofclass(OmlQuoteLiteral,l))
          -> convertQuoteLiteral(l),
        others
          -> return new JmlNullLiteral()
  end;


public convertNumericLiteral :
  OmlNumericLiteral ==>
  JmlNumericalLiteral
convertNumericLiteral(n) ==
  let val = n.getVal()
  in return new JmlNumericalLiteral(val);


public convertRealLiteral :
  OmlRealLiteral ==>
  JmlFloatLiteral
convertRealLiteral(n) ==
  let val = n.getVal()
  in return new JmlFloatLiteral(val);


public convertBooleanLiteral :
  OmlBooleanLiteral ==>
```

```
  JmlBooleanLiteral
convertBooleanLiteral(n) ==
  let val = n.getVal()
  in return new JmlBooleanLiteral(val);


public convertNilLiteral :
  OmlNilLiteral ==>
  JmlNullLiteral
convertNilLiteral(-) ==
  return new JmlNullLiteral();


public convertCharacterLiteral :
  OmlCharacterLiteral ==>
  JmlCharacterLiteral
convertCharacterLiteral(n) ==
  let val = n.getVal()
  in return new JmlCharacterLiteral(val);


public convertTextLiteral :
  OmlTextLiteral ==>
  JmlStringLiteral
convertTextLiteral(n) ==
  let val = n.getVal()
  in return new JmlStringLiteral(val);


public convertQuoteLiteral :
  OmlQuoteLiteral ==>
  JmlEnumLiteral
convertQuoteLiteral(n) ==
  let val = n.getVal()
  in return new JmlEnumLiteral(val);


end Vdm2Jml
```

B.2 Mapper from *JML* to *VDM++*

```
class Jml2Vdm


operations
```

```
public build_vdm :
  JmlSpecifications ==>
  OmlSpecifications
build_vdm(specs) ==
  let classes = specs.getClassList(),
      vdmclasses = convertJmlClasses(classes)
  in return new OmlSpecifications(vdmclasses);


public convertJmlClasses :
  seq of JmlWrappedJmlClass ==>
  seq of OmlClass
convertJmlClasses(s) ==
  return [convertJmlClass(s(i)) | i in set inds s];


public convertJmlClass :
  JmlWrappedJmlClass ==>
  OmlClass
convertJmlClass(c) ==
  let cl = c.getClassVal(),
      id = cl.getIdentifier(),
          ic = cl.getInheritanceClause(),
          ii = cl.getInterfaceInheritance(),
          ih = getInheritanceClauses(ic,ii),
          bd = cl.getClassBody(),
          body = convertClassBody(bd)
  in return new OmlClass(id,[],ih,body,false);


public getInheritanceClauses :
  [JmlClassInheritanceClause] *
  [JmlInterfaceInheritanceClause] ==>
  [IOmlInheritanceClause]
getInheritanceClauses(c,i) ==
  let s1 = getClassInheritance(c),
      s2 = getInterfaceInheritance(i)
  in if s1 = [] and s2 = []
         then return nil
         else return new OmlInheritanceClause(s1^s2);


public getClassInheritance :
  [JmlClassInheritanceClause] ==>
  seq of (seq of char)
getClassInheritance(c) ==
  if c <> nil
  then return [c.getIdentifierList()]
  else return [[]];
```

```
public getInterfaceInheritance :
  [JmlInterfaceInheritanceClause] ==>
  seq of (seq of char)
getInterfaceInheritance(c) ==
  if c <> nil
  then return c.getIdentifierList()
  else return [[]];


public convertClassBody :
  seq of IJmlDefinitionBlock ==>
  seq of IOmlDefinitionBlock
convertClassBody(s) ==
  let l =  [ convertJmlBlock(s(i)) | i in set inds s],
      q = removeNil(l)
  in return q;


public removeNil :
  seq of [IOmlDefinitionBlock] ==>
  seq of IOmlDefinitionBlock
removeNil(s) ==
  return [s(i) | i in set inds s & s(i) <> nil];


public convertJmlBlock :
  IJmlDefinitionBlock ==>
  [IOmlDefinitionBlock]
convertJmlBlock(b) ==
  cases true:
    (isofclass(JmlInstanceVariableDefinitions,b))
          -> return convertInstanceVariables(b),
    (isofclass(JmlValueDefinitions,b))
          -> return convertValueDefinitions(b),
    (isofclass(JmlInvariantDefinitions,b))
          -> return convertInvariantDefinitions(b),
    (isofclass(JmlOperationDefinitions,b))
          -> return convertOperationDefinitions(b),
        others
          -> return nil
  end;


------------------------------
-- Instance Variables --------
------------------------------


public convertInstanceVariables :
  JmlInstanceVariableDefinitions ==>
```

```
  OmlInstanceVariableDefinitions
convertInstanceVariables(s) ==
  let jml_vars = s.getJmlVariables(),
      java_vars = s.getJavaVariables(),
          vdm_1 = convertVariables(jml_vars),
          vdm_2 = convertVariables(java_vars),
          res = vdm_1 ^ vdm_2
  in return new OmlInstanceVariableDefinitions(res);


public convertVariables :
  seq of JmlVariable ==>
  seq of OmlInstanceVariable
convertVariables(s) ==
  return [ convertVariable(s(i)) | i in set inds s];


public convertVariable :
  JmlVariable ==>
  OmlInstanceVariable
convertVariable(v) ==
  let oldaccess = v.getAccess(),
      statickey = v.getStatickeyword(),
      access = buildAccessDefinition(oldaccess,statickey),
          id = v.getIdentifier(),
          tp = v.getType(),
          expr = v.getExpression(),
          assign = createOmlAssignmentDefinition(id,tp,expr)
  in return new OmlInstanceVariable(access,assign);


public buildAccessDefinition :
  JmlAccessDefinition *
  bool ==>
  OmlAccessDefinition
buildAccessDefinition(a,stk) ==
  let sc = a.getScope(),
      val = sc.getValue(),
          scope = buildScope(val)
  in return new OmlAccessDefinition(false,stk,scope);


public buildScope :
  nat ==>
  OmlScope
buildScope(n) ==
  cases true:
    (n = 0)
          -> return new OmlScope(0),
       (n = 1)
          -> return new OmlScope(1),
```

```
        (n = 2)
          -> return new OmlScope(3),
        others
          -> return new OmlScope(2)
  end;


public createOmlAssignmentDefinition :
  seq of char *
  JmlType *
  [JmlExpression] ==>
  OmlAssignmentDefinition
createOmlAssignmentDefinition(id,tp,expr) ==
  let newtp = convertType(tp),
      newexpr = convertExpression(expr)
  in return new OmlAssignmentDefinition(id,newtp,newexpr);


------------------------------
-- Values --------------------
------------------------------


public convertValueDefinitions :
  JmlValueDefinitions ==>
  OmlValueDefinitions
convertValueDefinitions(s) ==
  let l = s.getValueList(),
      q = convertValues(l)
  in return new OmlValueDefinitions(q);


public convertValues :
  seq of JmlValueDefinition ==>
  seq of OmlValueDefinition
convertValues(s) ==
  return [ convertValue(s(i)) | i in set inds s];


public convertValue :
  JmlValueDefinition ==>
  OmlValueDefinition
convertValue(v) ==
  let access = v.getAccess(),
      statkey = v.getStaticMod(),
          shape = v.getShape(),
          newaccess = buildAccessDefinition(access,statkey),
          newshape = convertValueShape(shape)
  in return new OmlValueDefinition(newaccess,newshape);
```

```
public convertValueShape :
  JmlValueShape ==>
  OmlValueShape
convertValueShape(s) ==
  let id = s.getIdentifier(),
      tp = s.getType(),
          ex = s.getExpression(),
          newtp = convertType(tp),
          newex = convertExpression(ex),
          pat = new OmlPatternIdentifier(id)
  in return new OmlValueShape(pat,newtp,newex);


------------------------------
-- Invariants ----------------
------------------------------


public convertInvariantDefinitions :
  JmlInvariantDefinitions ==>
  OmlInstanceVariableDefinitions
convertInvariantDefinitions(s) ==
  let l = s.getInvariantList(),
      q = convertInvariants(l)
  in return new OmlInstanceVariableDefinitions(q);


public convertInvariants :
  seq of JmlInvariantDefinition ==>
  seq of OmlInstanceVariableInvariant
convertInvariants(s) ==
  return [ convertInvariant(s(i)) | i in set inds s];


public convertInvariant :
  JmlInvariantDefinition ==>
  OmlInstanceVariableInvariant
convertInvariant(i) ==
  let expr = i.getPredicate(),
      newexpr = convertExpression(expr)
  in return new OmlInstanceVariableInvariant(newexpr);


------------------------------
-- Operations ----------------
------------------------------


public convertOperationDefinitions :
  JmlOperationDefinitions ==>
  OmlOperationDefinitions
```

```
convertOperationDefinitions(s) ==
  let list = s.getOperationList(),
      newlist = convertOperations(list)
  in return new OmlOperationDefinitions(newlist);


public convertOperations :
  seq of JmlOperationDefinition ==>
  seq of OmlOperationDefinition
convertOperations(s) ==
  return [ convertOperation(s(i)) | i in set inds s];


public convertOperation :
  JmlOperationDefinition ==>
  IOmlOperationDefinition
convertOperation(op) ==
  let access = op.getAccess(),
      stat = op.getStatickeyword(),
      newaccess = buildAccessDefinition(access,stat),
          id = op.getIdentifier(),
          t = op.getReturningType(),
          p = op.getParameterList(),
          tp = buildOperationType(t,p,id),
          params = buildParametersList(p),
          trl = op.getTrailer(),
          trailer = buildOperationTrailers(trl),
      shape = new OmlImplicitOperation(id,params,tp,trailer)
  in return new OmlOperationDefinition(newaccess,shape);


public buildOperationType :
  JmlType *
  seq of JmlParameter *
  seq of char ==>
  seq of OmlIdentifierTypePair
buildOperationType(t,s,id) ==
  let tp = convertType(t),
      ni = "var" ^ id
  in return [new OmlIdentifierTypePair(ni,tp) ];


public buildParametersList :
  seq of JmlParameter ==>
  seq of OmlPatternTypePair
buildParametersList(s) ==
  return [ buildParameter(s(i)) | i in set inds s];


public buildParameter :
  JmlParameter ==>
```

```
  OmlPatternTypePair
buildParameter(p) ==
  let id = p.getIdentifier(),
      tp = p.getType(),
          nt = convertType(tp),
      pa = new OmlPatternIdentifier(id)
  in return new OmlPatternTypePair([pa],nt);


public buildOperationTrailers :
  [JmlMethodSpecifications] ==>
  OmlOperationTrailer
buildOperationTrailers(specs) ==
  if specs = nil
  then return new OmlOperationTrailer(nil,nil,nil,nil)
  else let sp = specs.getSpecs(),
           al = specs.getAlso(),
           s1 = sp.getList(),
           s2 = al.getList(),
           s = s1 ^ s2
           in return buildTrailers(s);


--FIXME: if the operation has externals, it should be implicit
public buildTrailers :
  seq of IJmlOperationTrailer ==>
  OmlOperationTrailer
buildTrailers(s) ==
  let list = joinTrailers(s),
      pres = buildPreConditions(list),
          posts = buildPostConditions(list),
          extr = buildExternalConditions(list),
          excp = buildExceptionalConditions(list)
  in return new OmlOperationTrailer(extr,pres,posts,excp);


public buildPreConditions :
  seq of IJmlTrailers ==>
  [IOmlExpression]
buildPreConditions(s) ==
  let l = [ s(i) | i in set inds s &
          isofclass(JmlRequiresClause,s(i))]
  in return buildPreConditionsHelper(l);


public buildPreConditionsHelper :
  seq of JmlRequiresClause ==>
  [IOmlExpression]
buildPreConditionsHelper(s) ==
  if len s = 0 then return nil else
  let l = [ convertPreCondition(s(i)) | i in set inds s]
```

```
  in return landExpression(l);


public landExpression :
  seq of OmlExpression ==>
  IOmlExpression
landExpression(s) ==
  if len s = 1 then return hd s else
  let op = new OmlBinaryOperator(10),
      lhs = hd s,
          rhs = landExpression(tl s)
  in return new OmlBinaryExpression(lhs,op,rhs);


public buildPostConditions :
  seq of IJmlTrailers ==>
  [OmlExpression]
buildPostConditions(s) ==
  let l = [ s(i) | i in set inds s &
          isofclass(JmlEnsuresClause,s(i))]
  in return buildPostConditionsHelper(l);


public buildPostConditionsHelper :
  seq of JmlEnsuresClause ==>
  [IOmlExpression]
buildPostConditionsHelper(s) ==
  if len s = 0 then return nil else
  let l = [ convertPostCondition(s(i)) | i in set inds s]
  in return landExpression(l);


public buildExternalConditions :
  seq of IJmlTrailers ==>
  [OmlExternals]
buildExternalConditions(s) ==
  if len s = 0 then return nil else
  let l = [ convertAssignableClause(s(i))
                  | i in set inds s & isofclass(JmlAssignableClause,s(i)) ]
  in return new OmlExternals(conc l);


public buildExceptionalConditions :
  seq of IJmlTrailers ==>
  [OmlExceptions]
buildExceptionalConditions(s) ==
  if len s = 0 then return nil else
  let l = [ convertSignalsClause(s(i))
          | i in set inds s & isofclass(JmlSignalsClause,s(i)) ]
  in return new OmlExceptions(l);
```

```
public joinTrailers :
  seq of IJmlOperationTrailer ==>
  seq of IJmlTrailers
joinTrailers(s) ==
  return conc [ getTrailers(s(i)) | i in set inds s ];


public getTrailers :
  IJmlOperationTrailer ==>
  seq of IJmlTrailers
getTrailers(t) ==
  cases true:
    (isofclass(JmlBehaviourSpec,t))
          -> return getTrailerList(t),
        (isofclass(JmlExceptionalSpec,t))
          -> return getTrailerList(t),
        others
          -> return getTrailerList(t)
  end;


public getTrailerList :
  JmlBehaviourSpec ==>
  seq of JmlTrailers
getTrailerList(s) ==
  return s.getList();


public convertPostCondition :
  JmlEnsuresClause ==>
  OmlExpression
convertPostCondition(e) ==
  let expr = e.getEnsuresExpression(),
      nexp = convertExpression(expr)
  in return nexp;


public convertPreCondition :
  JmlRequiresClause ==>
  OmlExpression
convertPreCondition(e) ==
  let expr = e.getRequiresExpression(),
      nexp = convertExpression(expr)
  in return nexp;


public convertAssignableClause :
  JmlAssignableClause ==>
  seq of OmlVarInformation
convertAssignableClause(a) ==
```

```
  let l = a.getAssignableList(),
      mode = new OmlMode(0),
          nl = buildNames(l),
          s = buildVarInformation(nl,mode)
  in return s;


public buildNames :
  seq of (seq of char) ==>
  seq of OmlName
buildNames(s) ==
  return [ buildName(s(i)) | i in set inds s];


public buildName :
  seq of char ==>
  OmlName
buildName(s) ==
  return new OmlName(nil,s);


public buildVarInformation :
  seq of OmlName *
  OmlMode ==>
  seq of OmlVarInformation
buildVarInformation(s,m) ==
  let var = new OmlVarInformation(m,s,nil)
  in return [ var ];


public convertSignalsClause :
  JmlSignalsClause ==>
  OmlError
convertSignalsClause(s) ==
  let exc = s.getException(),
      prd = s.getPredicate(),
          npred = convertExpression(prd),
          id = getIdFromException(exc),
          expr = buildFalseExpression()
  in return new OmlError(id,npred,expr);


public getIdFromException :
  JmlException ==>
  seq of char
getIdFromException(e) ==
  return e.getIdentifier();


public buildFalseExpression :
  () ==>
```

```
  OmlSymbolicLiteralExpression
buildFalseExpression() ==
  let f = new OmlBooleanLiteral(false),
      expr = new OmlSymbolicLiteralExpression(f)
  in return expr;


------------------------------
-- Types ---------------------
------------------------------


public convertType :
  IJmlType ==>
  IOmlType
convertType(t) ==
  cases true:
    (isofclass(JmlCharType,t))
          -> return new OmlCharType(),
        (isofclass(JmlBoolType,t))
          -> return new OmlBoolType(),
        (isofclass(JmlEnumerationType,t))
          -> return convertEnumerationType(t),
        (isofclass(JmlIntegerType,t))
          -> return new OmlIntType(),
        (isofclass(JmlFloatType,t))
          -> return new OmlRealType(),
        (isofclass(JmlMapValueToValueType,t))
          -> return convertMapType(t),
        (isofclass(JmlSetValueType,t))
          -> return convertSetType(t),
        (isofclass(JmlSeqValueType,t))
          -> return convertSeqType(t),
        (isofclass(JmlObjectType,t))
          -> return new OmlTokenType(),
        (isofclass(JmlVoidType,t))
          -> return new OmlEmptyType(),
        (isofclass(JmlTupleType,t))
          -> return convertTupleType(t),
        (isofclass(JmlNatType,t))
          -> return new OmlNatType(),
        (isofclass(JmlClassType,t))
          -> return convertClassType(t),
        (isofclass(JmlClassName,t))
          -> return convertClassName(t),
        others
          -> return new OmlEmptyType()
  end;


public convertEnumerationType :
```

```
  JmlEnumerationType ==>
  OmlQuoteType
convertEnumerationType(enum) ==
  let lit = enum.getEnumLiteral(),
      newlit = convertLiteral(lit)
  in return new OmlQuoteType(newlit);


public convertMapType :
  JmlMapValueToValueType ==>
  OmlGeneralMapType
convertMapType(m) ==
  let domtp = m.getDomType(),
      rngtp = m.getRngType(),
          tpd = convertType(domtp),
          tpr = convertType(rngtp)
  in return new OmlGeneralMapType(tpd,tpr);


public convertSetType :
  JmlSetValueType ==>
  OmlSetType
convertSetType(s) ==
  let tp = s.getType(),
      newtp = convertType(tp)
  in return new OmlSetType(newtp);


public convertSeqType :
  JmlSeqValueType ==>
  OmlSeq0Type
convertSeqType(s) ==
  let tp = s.getType(),
      newtp = convertType(tp)
  in return new OmlSeq0Type(newtp);


public convertTupleType :
  JmlTupleType ==>
  OmlProductType
convertTupleType(t) ==
  let tps = t.getVals()
  in cases true:
       (len tps = 0)
             -> let t1 = new OmlEmptyType(),
                        t2 = new OmlEmptyType()
            in return new OmlProductType(t1,t2),
           (len tps = 1)
             -> let t1 = hd tps,
                        nt = convertType(t1),
                        t2 = new OmlEmptyType()
```

```
            in return new OmlProductType(nt,t2),
           others
             -> let t1 = hd tps,
                        lhs = convertType(t1),
                                rhs = convertRhsTypes(tl tps)
                    in return new OmlProductType(lhs,rhs)
    end;


public convertRhsTypes :
  seq of JmlType ==>
  IOmlType
convertRhsTypes(s) ==
  if len s = 1
  then let tp = hd s,
           nt = convertType(tp)
           in return nt
  else let tp = hd s,
           lhs = convertType(tp),
                   rhs = convertRhsTypes(tl s)
           in return new OmlProductType(lhs,rhs);


public convertClassType :
  JmlClassType ==>
  OmlCompositeType
convertClassType(c) ==
  let id = c.getId(),
          fl = c.getFieldList(),
          newfl = convertFieldList(fl)
  in return new OmlCompositeType(id,newfl);


public convertFieldList :
  seq of JmlField ==>
  seq of OmlField
convertFieldList(s) ==
  return [ convertField(s(i)) | i in set inds s];


public convertField :
  JmlField ==>
  OmlField
convertField(f) ==
  let id = f.getId(),
      tp = f.getType(),
          newtp = convertType(tp)
  in return new OmlField(id,newtp,false);


public convertClassName :
```

```
  JmlClassName ==>
  OmlTypeName
convertClassName(n) ==
  let name = n.getId(),
      newname = convertName(name)
  in return new OmlTypeName(newname);


------------------------------
-- Expressions ---------------
------------------------------


public convertExpression :
  IJmlExpression ==>
  IOmlExpression
convertExpression(e) ==
  cases true:
    (isofclass(JmlBracketedExpression,e))
          -> return convertBracketedExpression(e),
          (isofclass(JmlIfExpression,e))
          -> return convertIfExpression(e),
          (isofclass(JmlUnaryExpression,e))
          -> return convertUnaryExpression(e),
          (isofclass(JmlBinaryExpression,e))
          -> return convertBinaryExpression(e),
          (isofclass(JmlForAllExpression,e))
          -> return convertForAllExpression(e),
          (isofclass(JmlExistsExpression,e))
          -> return convertExistsExpression(e),
          (isofclass(JmlOldName,e))
          -> return convertOldName(e),
          (isofclass(JmlNewExpression,e))
          -> return convertNewExpression(e),
          (isofclass(JmlNewExpression,e))
          -> return convertName(e),
          (isofclass(JmlSetEnumeration,e))
          -> return convertSetEnumeration(e),
          (isofclass(JmlSequenceEnumeration,e))
          -> return convertSequenceEnumeration(e),
          (isofclass(JmlMapEnumeration,e))
          -> return convertMapEnumeration(e),
          (isofclass(JmlApplyExpression,e))
          -> return convertApplyExpression(e),
          (isofclass(JmlFieldSelectExpression,e))
          -> return convertFieldSelect(e),
          (isofclass(JmlLiteralExpression,e))
          -> return convertLiteralExpression(e),
          (isofclass(JmlBlockExpression,e))
          -> return convertBlockExpression(e),
          (isofclass(JmlThisExpression,e))
```

```
          -> return convertThisExpression(e),
          (isofclass(JmlInstanceOfExpression,e))
          -> return convertInstanceOf(e),
          others
      -> return new OmlUndefinedExpression()
  end;


public convertBracketedExpression :
  JmlBracketedExpression ==>
  OmlBracketedExpression
convertBracketedExpression(e) ==
  let expr = e.getExpression(),
      nexp = convertExpression(expr)
  in return new OmlBracketedExpression(nexp);


public convertIfExpression :
  JmlIfExpression ==>
  OmlIfExpression
convertIfExpression(e) ==
  let ifexp = e.getIfExpression(),
      thenexp = e.getThenExpression(),
          elseexp = e.getElseExpression(),
          nif = convertExpression(ifexp),
          nthen = convertExpression(thenexp),
          nelse = convertExpression(elseexp)
  in return new OmlIfExpression(nif,nthen,[],nelse);


public convertUnaryExpression :
  JmlUnaryExpression ==>
  OmlUnaryExpression
convertUnaryExpression(e) ==
  let op = e.getOperator(),
      ex = e.getExpression(),
          nop = convertOperator(op),
          nex = convertExpression(ex)
  in return new OmlUnaryExpression(nop,nex);


public convertOperator :
  JmlUnaryOperator ==>
  OmlUnaryOperator
convertOperator(op) ==
  let val = op.getValue() in
  cases true:
    (val = 4)
          -> return new OmlUnaryOperator(0),
        (val = 5)
          -> return new OmlUnaryOperator(1),
```

```
        (val = 10)
          -> return new OmlUnaryOperator(2),
        (val = 7)
          -> return new OmlUnaryOperator(3),
        (val = 2)
          -> return new OmlUnaryOperator(5),
        (val = 0)
          -> return new OmlUnaryOperator(6),
        (val = 6)
          -> return new OmlUnaryOperator(8),
        (val = 14)
          -> return new OmlUnaryOperator(9),
        (val = 12)
          -> return new OmlUnaryOperator(10),
        (val = 8)
          -> return new OmlUnaryOperator(11),
        (val = 15)
          -> return new OmlUnaryOperator(14),
        (val = 1)
          -> return new OmlUnaryOperator(15),
        (val = 13)
          -> return new OmlUnaryOperator(16),
        (val = 3)
          -> return new OmlUnaryOperator(17),
        others
          -> return new OmlUnaryOperator()
  end;


public convertBinaryExpression :
  JmlBinaryExpression ==>
  OmlBinaryExpression
convertBinaryExpression(e) ==
  let lhs = e.getLhsExpression(),
          op  = e.getOperator(),
          rhs = e.getRhsExpression(),
          nlhs = convertExpression(lhs),
          nop = convertBinaryOperator(op),
          nrhs = convertExpression(rhs)
  in return new OmlBinaryExpression(nlhs,nop,nrhs);


public convertBinaryOperator :
  JmlBinaryOperator ==>
  OmlBinaryOperator
convertBinaryOperator(op) ==
  let val = op.getValue() in
  cases true:
    (val = 1)
          -> return new OmlBinaryOperator(18),
        (val = 2)
```

```
        -> return new OmlBinaryOperator(23),
      (val = 3)
        -> return new OmlBinaryOperator(13),
      (val = 6)
        -> return new OmlBinaryOperator(22),
      (val = 7)
        -> return new OmlBinaryOperator(33),
      (val = 8)
        -> return new OmlBinaryOperator(1),
      (val = 10)
        -> return new OmlBinaryOperator(16),
      (val = 11)
        -> return new OmlBinaryOperator(0),
      (val = 12)
        -> return new OmlBinaryOperator(12),
      (val = 13)
        -> return new OmlBinaryOperator(4),
      (val = 14)
        -> return new OmlBinaryOperator(15),
      (val = 16)
        -> return new OmlBinaryOperator(28),
      (val = 19)
        -> return new OmlBinaryOperator(6),
      (val = 20)
        -> return new OmlBinaryOperator(34),
      (val = 21)
        -> return new OmlBinaryOperator(32),
      (val = 22)
        -> return new OmlBinaryOperator(2),
      (val = 23)
        -> return new OmlBinaryOperator(8),
      (val = 24)
        -> return new OmlBinaryOperator(30),
      (val = 25)
        -> return new OmlBinaryOperator(31),
      (val = 26)
        -> return new OmlBinaryOperator(14),
      (val = 27)
        -> return new OmlBinaryOperator(7),
      (val = 30)
        -> return new OmlBinaryOperator(21),
      (val = 31)
        -> return new OmlBinaryOperator(24),
      (val = 32)
        -> return new OmlBinaryOperator(19),
      (val = 33)
        -> return new OmlBinaryOperator(9),
      others
        -> return new OmlBinaryOperator()
end;
```

```
public convertForAllExpression :
  JmlForAllExpression ==>
  OmlForAllExpression
convertForAllExpression(e) ==
  let bind = e.getBindList(),
      expr = e.getExpression(),
          nexp = getQuantifierExpression(expr),
          nbind = convertBind(bind)
  in return new OmlForAllExpression(nbind,nexp);


public getQuantifierExpression :
  seq of IJmlExpression ==>
  IOmlExpression
getQuantifierExpression(s) ==
  if len s = 0
  then let lit = new OmlBooleanLiteral(true),
                   exp = new OmlSymbolicLiteralExpression(lit)
       in return exp
  else let fst = hd(s),
           lhs = convertExpression(fst),
           op = new OmlBinaryOperator(10),
           rhs = getQuantifierExpression(tl(s))
       in return new OmlBinaryExpression(lhs,op,rhs);


public convertBind :
  JmlQuantifierDeclaration ==>
  seq of IOmlBind
convertBind(q) ==
  let tp = q.getType(),
      ntp = convertType(tp),
          var = q.getVars(),
          pat = buildPatternIdentifiers(var),
          bind = new OmlTypeBind(pat,ntp)
  in return [bind];


public buildPatternIdentifiers :
  seq of (seq of char) ==>
  seq of IOmlPattern
buildPatternIdentifiers(s) ==
  return [ new OmlPatternIdentifier(s(i)) | i in set inds s];


public convertExistsExpression :
  JmlExistsExpression ==>
  OmlExistsExpression
convertExistsExpression(e) ==
  let bind = e.getBindList(),
```

```
      expr = e.getExpression(),
          nexp = getQuantifierExpression(expr),
          nbind = convertBind(bind)
  in return new OmlExistsExpression(nbind,nexp);


public convertOldName :
  JmlOldName ==>
  OmlOldName
convertOldName(o) ==
  let id = o.getIdentifier()
  in return new OmlOldName(id);


public convertNewExpression :
  JmlNewExpression ==>
  OmlNewExpression
convertNewExpression(e) ==
  let t = e.getType(),
      t1 = getTypeName(t),
          nt = convertName(t1),
          l = e.getExpressionList(),
          nl = convertExpressionList(l)
  in return new OmlNewExpression(nt,[],nl);


public getTypeName :
  IJmlType ==>
  JmlName
getTypeName(t) ==
  cases true:
    (isofclass(JmlClassName,t))
          -> return getClassName(t),
        others
          -> return new JmlName(nil,[])
  end;


public getClassName :
  JmlClassName ==>
  JmlName
getClassName(c) ==
  return c.getId();


public convertName :
  IJmlName ==>
  IOmlName
convertName(n) ==
  let cl = n.getClassIdentifier(),
      id = n.getIdentifier()
```

```
  in return new OmlName(cl,id);


public convertExpressionList :
  seq of [JmlExpression] ==>
  seq of OmlExpression
convertExpressionList(s) ==
  return [ convertExpression(s(i))
         | i in set inds s & s(i) <> nil];


public convertSetEnumeration :
  JmlSetEnumeration ==>
  OmlSetEnumeration
convertSetEnumeration(e) ==
  let l = e.getList(),
      q = convertExpressionList(l)
  in return new OmlSetEnumeration(q);


public convertSequenceEnumeration :
  JmlSequenceEnumeration ==>
  OmlSequenceEnumeration
convertSequenceEnumeration(e) ==
  let l = e.getList(),
      q = convertExpressionList(l)
  in return new OmlSequenceEnumeration(q);


public convertMapEnumeration :
  JmlMapEnumeration ==>
  OmlMapEnumeration
convertMapEnumeration(e) ==
  let ml = e.getList(),
      nm = convertMapletList(ml)
  in return new OmlMapEnumeration(nm);


public convertMapletList :
  seq of JmlMapLet ==>
  seq of OmlMaplet
convertMapletList(s) ==
  return [ convertMaplet(s(i)) | i in set inds s];


public convertMaplet :
  JmlMapLet ==>
  OmlMaplet
convertMaplet(m) ==
  let dval = m.getDomVal(),
      rval = m.getRngVal(),
```

```
          ndom = convertExpression(dval),
          nrng = convertExpression(rval)
  in return new OmlMaplet(ndom,nrng);


public convertApplyExpression :
  JmlApplyExpression ==>
  OmlApplyExpression
convertApplyExpression(e) ==
  let expr = e.getExpression(),
      expl = e.getExpressionList(),
          nexpr = convertExpression(expr),
          nexpl = convertExpressionList(expl)
  in return new OmlApplyExpression(nexpr,nexpl);


public convertFieldSelect :
  JmlFieldSelectExpression ==>
  OmlFieldSelect
convertFieldSelect(f) ==
  let expr = f.getExpression(),
      name = f.getName(),
          nexp = convertExpression(expr),
          newname = convertName(name)
  in return new OmlFieldSelect(nexp,newname);


public convertLiteralExpression :
  JmlLiteralExpression ==>
  OmlSymbolicLiteralExpression
convertLiteralExpression(e) ==
  let lit = e.getLit(),
      nlit = convertLiteral(lit)
  in return new OmlSymbolicLiteralExpression(nlit);


public convertBlockExpression :
  JmlBlockExpression ==>
  OmlLetExpression
convertBlockExpression(e) ==
  let bind = e.getBind(),
      expr = e.getReturnExpr(),
          nexp = convertExpression(expr),
          nbind = convertValueShapes(bind)
  in return new OmlLetExpression(nbind,nexp);


public convertValueShapes :
  seq of JmlValueShape ==>
  seq of OmlValueShape
convertValueShapes(s) ==
```

```
  return [ convertValueShape(s(i)) | i in set inds s];


public convertThisExpression :
  JmlThisExpression ==>
  OmlSelfExpression
convertThisExpression(-) ==
  return new OmlSelfExpression();


public convertInstanceOf :
  JmlInstanceOfExpression ==>
  OmlIsExpression
convertInstanceOf(i) ==
  let tp = i.getType(),
      ex = i.getExpression(),
          nt = convertType(tp),
          ne = convertExpression(ex)
  in return new OmlIsExpression(nt,ne);


------------------------------
-- Literals ------------------
------------------------------


public convertLiteral :
  IJmlLiteral ==>
  IOmlLiteral
convertLiteral(lit) ==
  cases true:
    (isofclass(JmlNumericalLiteral,lit))
          -> return convertNumericalLiteral(lit),
    (isofclass(JmlFloatLiteral,lit))
          -> return convertFloatLiteral(lit),
    (isofclass(JmlEnumLiteral,lit))
          -> return convertEnumLiteral(lit),
    (isofclass(JmlBooleanLiteral,lit))
          -> return convertBooleanLiteral(lit),
    (isofclass(JmlCharacterLiteral,lit))
          -> return convertCharacterLiteral(lit),
    (isofclass(JmlStringLiteral,lit))
          -> return convertStringLiteral(lit),
        (isofclass(JmlNullLiteral,lit))
          -> return new OmlNilLiteral(),
        others
          -> return new OmlNilLiteral()
  end;


public convertNumericalLiteral :
```

```
  JmlNumericalLiteral ==>
  OmlNumericLiteral
convertNumericalLiteral(lit) ==
  let val = lit.getVal()
  in return new OmlNumericLiteral(val);


public convertFloatLiteral :
  JmlFloatLiteral ==>
  OmlRealLiteral
convertFloatLiteral(lit) ==
  let val = lit.getVal()
  in return new OmlRealLiteral(val);


public convertEnumLiteral :
  JmlEnumLiteral ==>
  OmlQuoteLiteral
convertEnumLiteral(lit) ==
  let val = lit.getVal()
  in return new OmlQuoteLiteral(val);


public convertBooleanLiteral :
  JmlBooleanLiteral ==>
  OmlBooleanLiteral
convertBooleanLiteral(lit) ==
  let val = lit.getVal()
  in return new OmlBooleanLiteral(val);


public convertCharacterLiteral :
  JmlCharacterLiteral ==>
  OmlCharacterLiteral
convertCharacterLiteral(lit) ==
  let val = lit.getVal()
  in return new OmlCharacterLiteral(val);


public convertStringLiteral :
  JmlStringLiteral ==>
  OmlTextLiteral
convertStringLiteral(lit) ==
  let val = lit.getVal()
  in return new OmlTextLiteral(val);


end Jml2Vdm
```

Appendix C

Input/Output Stream

In this appendix, the specification of the case study Input/Output Stream is presented. Each section contains the specification and the results of applying the mapper, of one class. Thus, section C.1 contains the *VDM++* specification of the *IO* class and the resulting *JML* specification, after applying the mapper. The same happens for each of the other classes.

C.1 IO

C.1.1 Specification

```
class IO

types

public byte = nat;

operations

public read : seq of byte  ==> nat
read(b) == is subclass responsibility;

public write : seq of byte  ==> ()
write(b) == is subclass responsibility;

public close : () ==> ()
close() == is subclass responsibility;

end IO
```

C.2 InputStream

C.2.1 Specification

```
class InputStream is subclass of IO

operations

public read : seq of byte ==> nat
read(b) == is not yet specified;

public close : () ==> ()
close() == is not yet specified;


end InputStream
```

C.3 OutputStream

C.3.1 Specification

```
class OutputStream is subclass of IO

operations

public write : seq of byte ==> ()
write(b) == is not yet specified;

public close : () ==> ()
close() == is not yet specified;

end OutputStream
```

C.4 IOStream

C.4.1 Specification

```
class IOStream is subclass of InputStream, OutputStream

operations

public read : seq of byte ==> nat
```

```
read(b) == is not yet specified;

public write : seq of byte  ==> ()
write(b) == is not yet specified;

public close : () ==> ()
close() == is not yet specified;

end IOStream
```

Terminology

Abstract Class An abstract class is an *abstract type* containing variables and methods (with or without bodies), whose purpose is to act as a place-holder for additional variables and methods to be added by subclasses.
It is not possible to instantiate abstract classes. Furthermore, it is not necessary to have a complete implementation of it, leaving the implementation details to a class that *extends* the abstract class. Thus, it is possible for each subclass to override the defined methods in the abstract class specializing them according to their own needs.
Moreover, the abstract classes specify a public interface which can be inherited by its subclasses, allowing all the subclasses to have the same interface.
Finally, Java does not allow multiple inheritance concerning to abstract classes. A non-abstract class can only extend one abstract class, and an abstract class can be extended by a number of classes.

Abstract Syntax Tree Is a tree representation of the syntax of a given source code. After parse a given input file with a parser, an AST is created with the syntactic information of the input file.

Class Unlike the abstract class, a class, composed by variables and methods, must provide a complete implementation of their own methods.
Furthermore, a class is meant to have objects instantiated from it, providing those objects has the same variables and methods, and letting them have their own values.
As it was explained above, a class can implement a number of Interfaces and only extend one class/abstract class. For each Interface implemented, the class must complete its methods and the same happens for the extended abstract class.
Furthermore, it is also possible to extend one non-abstract class, and re-implement

all its methods.

Interface An Interface is conceptually an $^{\tau}$*abstract type* composed simply by method headers and static final variables (constant declarations), whose purpose is to specify how the interconnection between different systems and the one in question should work.
Furthermore, the implementation of the methods and the creation of non-static final variables is forbidden in an Interface, and should be left for an implementation class.
Each class (including abstract classes) can implement a number of interfaces, and it should specify the methods presented in each one. In fact, this is the only kind of multiple inheritance allowed in Java.

Object-Oriented paradigm Is a programming paradigm that makes use of objects are a mean of interact with design applications and computer programs.

Pure Type Pure types are immutable types, side-effect free. This means that after the creation of an object of a pure type, its value cannot be changed in any state of the execution of a program.

List of Figures

List of Tables