Connecting between VDM++ and JML

Carlos M. G. Vilhena

Engineering College of Århus,
Dalgas Avenue, DK-8000 Århus, Denmark

Abstract. This paper discusses 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. The development of a prototype proof-of-concept implementation for this bi-directional conversion is being carried through. At a later stage this prototype will be integrated on top of the *Eclipse* [8] platform as part of the *Overture Tool* [9].

1 Introduction

In order to establish a bidirectional connection between *VDM++* and *JML* it is necessary to explain its purposes and limitations, as well as identifying the potential advantages this may bring to software development. These are as follows: From a tool support perspective, it will be possible to take advantage of tool support available for both *VDM++* and *JML* sides by moving the specifications from one side to the other.

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.

However, such a connection has limitations that must be taken into account. There are semantic differences between *VDM++* and *JML*, namely in the way some typical Object-Oriented features and constructs semantics. These limitations are presented in section 2. A conceptual analysis is shown in section 3. The correlation between *VDM++* and *JML* is present in section 4. Furthermore, a case study is presented in section 5. An analysis of this connection is shown in section 6 and finally in section 7 a conclusion and future work can be found.

2 Limitations

Although *VDM++* and *JML* are Object-Oriented specification languages, they have semantic differences concerning some features such as inheritance. *VDM++* allows multiple class inheritance. On the other hand, *JML* as a specification language for *Java*, does not allow multiple class inheritance; only multiple interface inheritance. This would become a problem when one wants to move a *VDM++* specification with multiple class inheritance to *JML*. In this case, the adopted solution, as in VDMTools [4], will be ask the user in the connection process which classes will become interfaces and which class will remain as superclass. However, the transformation into interfaces can only be made under special circumstances, due to the fact that an interface has special rules such as the absence of bodies in methods and constructors, among others [11]. Furthermore, specification inheritance in *JML* guarantees that a subclass specification is stronger than the superclass specification. When it comes to method specifications, a pre-condition of an overriding method can only be weaker than the pre-condition of the overriden method. Moreover, the post-condition of an overriding method can only be stronger than the post-condition of the overriden method [10].

In *VDM++*, specification inheritance is not currently being considered. Thus, an overriding method will not inherit the specifications of the overridden method. This means that when one is mapping from *JML* to *VDM++* the semantic meaning of specification inheritance will be lost.

There are other limitations such as visibility issues and lack of semantic meaning of such constructs both in *JML* and *VDM++*. These are minor limitations which are explained in detail in [11].

3 Conceptual analysis

Although this connection is intended to connect *VDM++* and *JML*, it is not possible to separate *JML* from *Java* because *JML* is a behavioural specification language for *Java* modules. This proximity between *JML* and *Java* leads to some decisions to be made in order to build a proper connection. The major decisions to be made were:

- Inheritance vs Refinement approach;
- Usage of *JML* pure types or *Java* types.

Regarding the first item, there was a need of deciding whether to use an inheritance approach or a refinement approach as a specification approach for *JML*. Since *JML* assertions can be written inside *Java* classes, abstract classes or interfaces, and since the goal of this connection is to connect *VDM++* and *JML* leaving apart the *Java* implementation of the methods and constructors for a future *Java* code generator, it was important to decide where and how the *JML* assertions should be written when moving from *VDM++* to *JML*. After some considerations explained in detail in [11], it was possible to understand that the inheritance perspective was not a good approach.

If *JML* assertions were written either in a class or abstract class, when one has an implementation of it, one should connect the generated *JML* file with the implementation by means of inheritance, *i.e.*, extending the implementation. This means that if one wants to extend another class it would not be possible. On the other hand, if *JML* assertions were written inside *Java* interfaces, it would not be possible to connect constructors, because interfaces do not allow the usage of them. This means that if there are assertions related to constructors (e.g. pre-conditions), they could not be written at the *JML* level.

The refinement approach consists in writing *JML* assertions apart from the implementation. Those specification files are similar to *Java* files (they can also be classes, abstract classes or interfaces), however their usage is different. When one wants to connect an implementation with the specification, one should use the *JML* keyword *refines* followed by the correspondent file where the specification is. This way, the specification present in the pointed file will be connected to the implementation, and the assertions can be checked at runtime. The refinement approach only offers limitations regarding interfaces, for the same reasons presented above. The other two possibilities, which are classes or abstract classes, are equivalent in a sense they do not offer any limitation. For a question of semantic equivalence with *VDM++*, classes were chosen to sustain *JML* assertions.

With respect to the types, a decision was carried through. As a result of *JML* assertions only admit the usage of pure types and operators [7], *JML* pure types were chosen to map with *VDM++* types due to the fact that they are pure types, *i.e.*, their usage cannot cause side effects at runtime. A deep study of the correlation between *VDM++* and *JML* types and their associate constructs has been made in [11].

4 Correlation between *VDM++* and *JML*

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:

- **Types**: Since both *JML* and *VDM++* have pre-defined types, the correlation between those types, and associated constructs, must be analysed, in order to correctly proceed to the mapping between them;
- **Constructs**: The languages constructs should also be correctly mapped in order to maintain the semantic meaning.

4.1 Types

With respect to types, while *JML* offers a large number of types, *VDM++* is more restrictive. If one wants to use a given *JML* type that has no correspondence

at the *VDM++* side, either the user must be advised during the transformation or the user is not allowed to use a specific type. Although most of the types from *VDM++* are directly mapped into *JML* types, due to their semantic similarities, there is one particular type, the composite type, that has no direct correspondence to a *JML* pure type. A composite type is equivalent to a record type. It can have a number of components of a specific type, and it is possible to access directly to a component by means of an identifier. Even so there is no specific correspondence with a *JML* pure type, these types can be mapped into *Java* classes. Each of their components can be seen as instance variables of that class, and if the type has a specific invariant associated, that invariant can be converted at the *JML* side to an instance invariant (see [11]).

Besides the referred issue, each type has its own pre-defined constructs that allows one to manipulate an object of that type. As in the previous situation, each *JML* type has more associated constructs than each *VDM++* type. In this situation, if a given type A at the *JML* side is considered semantically identical to a type B at the *VDM++* side, then each construct that has no match both from *VDM++* and *JML* should be defined apart in a specific library to be used together with the users specifications. This way, although not all types are connected due to some limitations, at least the constructs of the mapped types are present for the user to use.

4.2 Constructs

For the purposes of the proposed connection between *VDM++* and *JML*, a small number of constructs will be considered from both specification languages. The reason behind this is that this project targets the exploration of a subset of both languages where a connection is possible and to build a prototype proof-of-concept implementation of it. Concerning all the other constructs, it will be possible to add them as an extension of the referred connection, in a point in time.

The considered constructs are pre-conditions, post-conditions, exceptional conditions, external conditions and invariants (see [2] for *VDM++* and [6] for *JML*). All of these constructs have a correspondent construct in *VDM++* and *JML*, however with a number of semantic differences. There is a first complication concerning specification inheritance. Although *VDM++* is an object-oriented specification language, it does not consider specification inheritance. On the other hand, specification inheritance in *JML* is well established, *i.e.*, it is only possible to weaken the pre-condition or strength the post-condition of an overriding method [10].

Despite the referred difference, both *VDM++* pre- and post-conditions are similar to the *JML* requires and ensures clauses, respectively [11]. Concerning the exceptional conditions, which are meant to deal with exceptional behaviour, have also a number of semantic differences in *VDM++* and *JML*. *JML* allows one to specify under which conditions a specific exception may possible be thrown [6]. However, in *VDM++* there is no semantic equivalence to exceptions. The construct responsible for dealing with exceptional behaviour, the *errs* clause,

defines exactly under which condition an error can occur and what are the consequences for the result of calling the operation. Each condition defined using the referred construct has a pre-condition, which represents a possible error situation, and a post-condition, which represents the consequence for violating the associated pre-condition (see [11]).

Concerning the external conditions, *JML* only allows one specific operation to be *pure* if the operations is side-effect free (see [1], [7], [6]). In order to have this guarantee, the operation should explicitly indicate which variables will be accessed using the *assignable* clause. Although the semantic meaning of *purity* is not present in *VDM++*, it is also possible to indicate which instance variables will be accessible, however, in a different way. It is only possible to use this construct, named *ext*, within implicit operations definition.

Finally, the invariants will also be considered within the proposed connection. In *VDM++*, it is possible to define invariants both for types and instance variables. On the other hand, in *JML* it is possible to define both static and instance class invariants [6].

There are a number of semantic differences between invariants in *VDM++* and in *JML* [11]. *VDM++* invariants do not have visibility constructs associated, *i.e.*, it is possible for example to use both private and public variables in an instance invariant. On the other hand, *JML* invariants can have explicit visibility constructs associated. Furthermore, *VDM++* allows one to define type invariants, in order to limit the possible values of the type. Concerning *JML*, it is only possible to define class invariants. If one uses a pre-defined *VDM++* type which is associated to a pre-defined *JML* type, the invariant defined at the *VDM++* side will not have proper equivalence in *JML*. However, if one defines a compound type in *VDM++*, which is mapped into a class in *JML*, then the type invariant will be semantically identical to a class invariant in *JML*.

For more details about the mapping between these constructs, see [11].

5 Case study: Alarm System

In order to exemplify how should the proposed connection work, a case study was chosen: the alarm system of a chemical plant [5]. It consists in a system that manages the calling out of experts to deal with operational faults discovered in a chemical plant. There are three entities in this system: expert, alarm and plant. Each expert has a number of qualifications that allows one to solve problems within ones areas of expertise. Furthermore, the alarm represents the a signal that must be sent to a specific expert whose area of expertise matches the problem in the chemical plant. Thus, whenever an alarm is raised, the expert on duty with the required expertise must be called.

For the purpose of this paper, only the *VDM++* and the *JML* model of the expert will be presented. However, the complete case study can be seen in [11]. The *JML* specification uses the refinement approach proposed by this connection between *VDM++* and *JML*. Thus, one implementation should be

connected with this specification by means of the *refine* keyword. Moreover, it also takes advantage of the *JML* pure types, invariants and pre-conditions.

```
//@ 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();
}
```

There are some information missing comparing with the *VDM++* model. All the constructors and methods bodies from the *VDM++* specification will no appear at the *JML* side, because the main goal of the proposed conneciton between *VDM++* and *JML* is to connect specifications, leaving the implementations for the user or for a future *Java* code generator.

class $Expert$
instance variables
$quali : Qualification$-set;
inv $ExpertInv(quali)$

functions

$ExpertInv : Qualification$-set $\rightarrow \mathbb{B}$

$ExpertInv(s) \triangleq$
$s \neq \{\}$

types

public $Qualification = \text{MECH} \mid \text{CHEM} \mid \text{BIO} \mid \text{ELEC}$

operations
public

$Expert : Qualification$-set $\xrightarrow{o} Expert$

$Expert(qs) \triangleq$
$quali := qs$
pre $ExpertInv(qs)$;

public

$GetQuali : () \xrightarrow{o} Qualification$-set

$GetQuali() \triangleq$
return $quali$

end $Expert$

6 Tool structure: considerations

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 (see [3]). After this process, the *Java* classes will be gathered in an eclipse plugin and the connection will finally be built.

Conceptually, this will be a bidirectional connection between *VDM++* and *JML*. It will make a strong use of syntactic analisys 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 assign semantic value to the input sequence. As the parser reads token input sequences, it will build a structure suitable for further treatment. This structure is known as Abstract Syntax Tree (AST). However, this AST must be built depending on the meaning of 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. This structure and the associated building process are explained in subsection 6.1 and they compose the first component of this connection.

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 compose the second component of this connection and it is explained in detail in subsection 6.2.

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. The pretty-print of the ASTs corresponds to the third and final component of this connection. However, it will not be explained in this paper due to its simplicity.

Concerning the user interaction with this connection, it is expected that the user interacts with it whenever the user wants to move a specification from one specification language to another. At this point, the user will be briefed with a *log* file and eventually some decisions to be made manually (details in subsection 6.2).

6.1 Parser and AST generation component

This component should be partially specified and partially implemented outside the specification. The reason behind this statement is 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.

- 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.

  At this point, it is possible to parse a number of *VDM++* files and retrieve the correspondent ASTs using the Overture parser. Thus, there is no need to specify this part of the first component.
- At the *JML* side, the forth version of the *JML* tools will be used for the sake of simplicity and compatibility. A *JML* parser is included on this set of tools, and it 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 comprehends 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 *JML*. This process evolves the use of a tool designed for the Overture project. The tool is called *ASTGEN* and from
  types representing a language it generates *Java* classes representing those types. Thus, the first step evolves the specification of an abstract representation of *JML*, *i.e.*, a representation with no *syntactic sugar*, by means of types. In this file, 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.

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 correspondent 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 type inside the abstract syntax file representing *JML*;
- In the ASTs converter, one will change the visitor in order to write the correct construct in the right place instead of null.

In short, in this step the abstract syntax of *JML* will need to be specified. Concerning all the other steps explained above, they will be implemented in *Java*.

6.2 Mapper component

After completing the previous steps, 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 relevant information, that information should be converted to other ASTs representing the target language. However, such conversion should respect the semantic rules defined in this paper, 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, *i.e.*, one should convert specifications as their needs. However, there will be a component to control such conversions in order to interact with the user for the following purposes:

- to inform the user about potential constructs being used that are not being considered by the mapper;
- to inform the user about potential semantic losses when moving from one side to another;
- to ask the user assistance in making decisions, if needed.

This entity will work both as a *log* file of the connection and also as an enquire to the user in order to make decisions. This is an important step of the mapper due to the fact that this connection will have limitations, thus the user must be informed about them and, if needed, interview.

In order to ease the user interaction with this connection, a *Graphical User Interface* (GUI) should be designed. However, this will certainly be apart from the specification. As a consequence of the specification being code generated to *Java* in a future stage, the GUI should also be designed directly in *Java*.

7 Conclusion and future work

The first goal of this work was to explore the possibilities for automatic translation between a subset of *VDM++* and *JML*. This exploration was carried through and it became clear that this connection is possible and, most important of all, it makes sense. Connecting these two specification languages will allow one to use tool support from both sides and when a future *Java* code generator becomes available for the Overture it will be possible to connect both specifications and implementations between *VDM++* and *Java* with *JML* assertions.

Regarding the future work, the most important step is to finish the implementation of this connection. Furthermore, the extension of it should be considered in order to enlarge the subsets of *VDM++* and *JML* being considered. With a presence of a *Java* code generator as in VDMTools, some work should be done connecting it to this mapper in order to be possible the automatic generation of both assertions and implementations. Finally, a graphical user interface is also an important step to perform in order to make this mapper more user-friendly.

References

1. P. Muller A. Darvas. Reasoning about method calls in *JML* specifications. In *Formal Techniques for Java-like Programs*, 2005.
2. CSK Corporation. *The VDM++ Language*. CSK, 2005.
3. CSK Corporation. *The VDM++ To Java Code Generator Language*. CSK, 2007.
4. CSK. VDMTools homepage. *http://www.vdmtools.jp/en/*, 2007.
5. John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs for Object–oriented Systems*. Springer, New York, 2005.
6. et al Gary T. Leavens. Jml reference manual. Available from *http://www.jmlspecs.org*, October 2007.
7. 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.
8. Jim D'Anjou Sherry Shavor Scott Fairbrother Dan Kehn John Kellerman Pat McCarthy. *The Java developer's guide to Eclipse*. Addison-Wesley Professional, May 2003.
9. The overture project homepage. *http://www.overturetool.org*, 2008.
10. Gary T. Leavens Patrice Chalin, Joseph R. Kiniry and Erik Poll. Beyond assertions: Advanced specification and verification with jml and esc/java2. pages 342–363, 2006.
11. Carlos M. G. Vilhena. Connecting between *VDM++* and *JML* (forthcomming), 2008.