Engineering College of Århus

Master of Computing Science Thesis

VDM++ Test Automation Support

by

Adriana Sucena Santos

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

Aarhus, 2008

Abstract

Testing is an important, expensive, repetitive and exhaustive task in software development. It does not guarantee that any kind of model has no errors, however the developer can be more confident that a model is working properly after testing it. If the testing process is done along the development, it will have to be frequently repeated because a small change in the model might influence its behaviour. Repeating tests, without automation support, each time a change is made is a tedious manual task. It is also time consuming and, consequently, expensive.

In this thesis, it is suggested theoretical and practical approaches of test automation in VDM++. The main goal is to contribute to reduce the effort VDM++ users need to spend to test VDM++ models.

Acknowledgements

I would like to thank Professor Peter Gorm Larsen who gave me a crucial professional and personal support during my Master thesis. He was also a career advisor and he was always making efforts in order to improve my knowledge and abilities as a researcher and as a possible future Software Developer in a company. Having Professor Peter Gorm Larsen has supervisor is an remarkable experience.

I also would like to thank Marcel Verhoef, who volunteered to extend the Overture Parser, as it was crucial for my thesis.

Furthermore, I would like to thank Professor Luís Soares Barbosa for his support before comming to Denmark.

Finally, Carlos Vilhena gave me the straights I very much needed when my work was not going as well as I was expecting. He was also always updated with my thesis and available to discuss with me the decisions I had to make, or provide me technical information where he was more comfortable than me. Thank you.

Contents

Chapter 1

Introduction

This chapter provides a background and introduce the context of this thesis, in section 1.1, where the main subjects that led this thesis are described. Thus, the formal modelling language VDM++ is described, followed by the Overture Project and the analysed testing strategies. The aim of this thesis is provided in section 1.2, the expected outcomes in section 1.3 and finally an outline of the overall thesis structure in section 1.4.

1.1 Background

This section introduces the main subjects that led this thesis. Thus, details of the VDM++ modelling language relevant for this thesis are described in subsection 1.1.1. The Overture Project is described in subsection 1.1.2. An extension of the Overture Parser was developed and thus a description of the tool support in this context is provided in subsection 1.1.3. Finally, the testing strategies analysed in this thesis are described in subsections 1.1.5 and 1.1.4.

1.1.1 VDM++

VDM++ is a formal modelling language, an extension of VDM-SL (Vienna Development Method Specification Language), handling object-oriented structuring and concurrency.

In formal software developments, a specification is a collection of properties that a system is supposed to satisfy. An implementation is the materialization of a specification enforcing its properties while seeking its efficient encoding in a particular programming language or framework. Thus, specifications are more abstract than implementations and the relationship between specifications and implementations is one-to-many. In context of this thesis, a specification is an

implicit style VDM++ model and an implementation is an explicit style VDM++ model.

VDM++ can be beneficial in software design because the use of this modelling language provides a rigorous description of a desired functionality. This rigorous description is possible due to the possibility of formalising properties as invariants on instance variables and pre and postconditions on operations which will last until the end of the development of the model. VDM++ can also be beneficial in testing because a model can either be a source for possible tests or an oracle against which test outcomes can be assessed [9].

A small description of the features becessary to understand this thesis are described below. For a complete coverage of VDM++, the reader is referred to the VDM++ language manual [7] or the VDM++ book [9].

A VDM++ model is structured with a collection of classes which can be connected by inheritance and association relationships. The inheritance is identified in VDM++ with the keywords *is subclass of*. Thus, a super class called *SuperClass* and a subclass of *SuperClass* called *SubClass* would be specified with the following:

```
class SubClass is subclass of SuperClass
```

Each class can has various defined blocks. The blocks we make use of are instance variables and operation definitions.

Instance variables correspond to state variables. In other words, they describe attributes which belong to every object instantiated from a class. Each instance variable has an associated type and they are allowed to have one of three accessibility levels: *public*, *private* or *protected*, as it is exemplified below.

```
instance variables

public var1 : nat := 2;
private var2 : nat := 2;
protected var3 : nat := 2;
```

A *public* variable can be used by any class, without restrictions. A *private* variable can only be used by its own instance of the class. Finally, a *protected* variable can only be used by the subclasses of its class. If none of these keywords is used for an instance variable, it will be defined as *private*, by default value.

An operation defines a possible behaviour of objects from a class. It can receive input parameters and retrieve a result. The accessibility levels of operations are analogous with the ones defined for the instance variables. Operations can be used to update the value of an instance variable. An example is provided below, where the operation *change* change the value of the instance variable

var1 from 2 to 4 the first time it is called.

```
instance variables

public var1 : nat := 2;

operations

public change : () ==> ()
change() == var1 := 4;
```

This introduced the basic concepts in VDM++. However, the reader is assumed to be comfortable with the VDM++ language in order to understand the models presented along the thesis.

1.1.2 The Overture Project

This Master thesis is incorporated in the Overture project [24] which is a community based project aiming to develop a comprehensive tool set for VDM++ on top of the Eclipse platform [18]. Overture builds on a long history of research and industrial application based on VDM.
Once a month, the Overture community met in a net meeting and, along many other subjects, the current status of this thesis was presented and discussed by the community. The feedback from the community enriched this project by being possible to develop a test automation support tool which is closer to what the users are expecting to have.

For the Overture Project, a specification language named Overture Modelling Language (OML) was created. This OML is an extension of the VDM++ specification language used by VDMTools. Furthermore, a parser was created in order to syntatically process OML specifications. In subsection 1.1.3, a description about the tool support used in this context is provided.

1.1.3 The Overture Parser

An extension was made in the Overture Parser, for the combinatorial testing tool developed in this thesis, described in 4.

The first stage of the Combinatorial Testing tool is to scan and parse the model given by the tester. The traces from where test cases are automatically generated must respect a defined grammar which is provided in sub-section 4.3. In order to scan and parse the model, the following tools were used:

- ASTGEN: this tool is responsible for collecting a set of VDM++ types and for transforming them into Java classes representing those types.
- Jflex is a lexical analyser responsible for converting a sequence of input characters into a sequence of categorized blocks of text, called tokens.

- Byaccj is a parser generator for Java. It is responsible for reading an input file and, together with a scanner, it generates a parser based on the information present in the production rules.

The output given by the ASTGEN tool are both the Java classes and the corresponding VDM++ classes representing the VDM-SL types written in the input file. Within the development of the Byaccj file, in the right hand side of the production rules, Java constructors are used, from the Java classes generated by the ASTGEN, to create objects matching the correspondent production rule. For each construct defined as a VDM-SL type in the input file, a Java class is generated. Furthermore, when the parser match that specific type in a production rule, an object of the created Java class is created in order to hold the relevant information. Byaccj also needs the result given by Jflex, which correspond to the left hand side of the production rules [12].

Combining these three tools, the outcome of parsing an input file respecting the defined trace is an abstract syntax tree (AST) containing all the relevant information parsed from the input file. This AST can be seen as an intermediate structure ready for further treatment, such as pretty-printing in order to obtain the concrete syntax of the parsed language.

1.1.4 Black Box Testing Strategies

Black Box Testing is also called *Functional Testing*, *Opaque Testing* and *Closed Box Testing* in the testing literature. It is a testing strategy focused on determining whether or not a program does what it is supposed to do, based on its functional requirements. It does not need any knowledge of the internal workings, like the internal design or code structure, of the item being tested [4].
When this strategy is applied, the tester only has information about the *legal* inputs and what the expected outputs should be, but not how the program actually reaches those outputs. Figure 1.1.4 illustrates this description. This means that

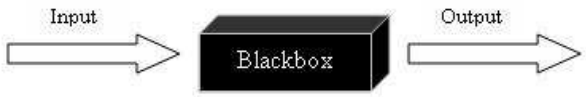

Figure 1.1: Black Box Testing

the tester and the programmer can be independent of each other and the test planning can begin as soon as the specifications are written. The tester does not need to know how the program is implemented or how to deal with all the features evolved in the program development. Actually the tester should not know anything about the implementation because that could influence the testing.

It is hard to apply this strategy if there is not a clear and concise specification. Another disadvantage is that many paths of the program will not be covered during the testing. Some potentially complex segments of code cannot be tested directly and that is another disadvantage, because the probability of finding errors in complex segments is normally higher than in simpler segments [14].
Combinatorial Testing is a Black Box Testing method which is described below, and was used for developing a tool in this thesis.

Combinatorial Testing The Combinatorial Testing strategy consists on using models to generate the minimum number of test inputs in order to cover all the selected combinations of input values [6].

Thus, test data is generated automatically from models developed by testers. In the implementation of this strategy, it is important to filter the irrelevant test data in order to reduce the possibility of having combinatorial explosion. Even so, the tester must carefully design the model to also manage combinatorial explosion issues.

The main interest of combinatorial testing for this thesis is to automatically generate test sequences based on chosen operations from a model and chosen inputs. An algorithm will be used in order to filter the irrelevant test sequences, as it is described in 4.4.

Domain Testing

Domain Testing is the most frequently described test technique [14] in the literature.

Models with infinite input types cannot be tested with all possible inputs. Even when the input type is finite, it is frequently not possible to test a model for all the cases, due to the large number of possibilities. However, it is possible to divide the domain in finite partitions, where the values of each partition should be equivalent, in some way. This consists on the testing strategy called Domain Testing. To choose the partitions, the boundary values should be taken in consideration. Each boundary value is a good candidate to create two partitions.

The tester should choose the best representatives of each partition and only test that values; the other inputs are supposed to have the same behaviour. Unfortunately, the behaviour is not always similar and, consequently, errors that are not at boundaries or in obvious special cases will not be detected [5]. This represents a disadvantage of Domain Testing.

This testing strategy is applied in the VDM++Tesk tool, as it is described in subsection 2.1.2. It is also suggested as future work of the combinatorial testing tool developed in this thesis, in section 6.3.

Regression Testing

The term *Regression Testing* is used in two different situations. In both of them, the idea is to reuse old tests [4].
One application of this method is when a bug is found with a *test A*, it is fixed and then the *test A* is repeated. The objective is to be sure that the fix really fixed the bug.
Another application of this method is when the tester wants to certificate that the code which fixed a bug did not create new bugs.
Both types of tests should always be applied when a bug is fixed.
It should be noticed that when a user inform the developer about a bug, many scenarios can happen: the bug can be fixed and more bugs were created or not because of that; the user could had explained in a wrong way so that the developer understood that the error was another one and so the bug is not fixed; the user could had explained extremely well which was the error, but the developer made a bad interpretation, so he did not fix the problem. In these last two situations, new bugs could also appear after the changes made by the developer.

Regression Testing is related to the combinatorial testing tool developed in this thesis because the test cases which are automatically generated are saved and can be reused in the future by the developer, after updating the model.

1.1.5 White Box Testing Strategies

The White Box Testing is also called *Glass Testing*, *Structural Testing*, *Open Box Testing* and *Clear Box Testing*.

It is a testing strategy focused on determining whether or not a model does what it is supposed to do based on the defined requisites. To apply this strategy, it is necessary to have an intimate knowledge of the source code and all the features evolved in the development of the model. Thus, this strategy is often applied by the developers of the model [27]. If it is not used by a developer, then the tester must start to comprehend and analyze the model before start testing. This is a disadvantage comparing to the Black Box Testing strategy.
An advantage of this strategy is that if a bug is detected, it won't be hard to identify the specific lines of the model involved. It also allows to directly test specific complex segments of the model, where the tester might think there will be a higher probability of finding bugs.
Mutation Testing is a White Box Testing method and it is described below. A theoretical approach of this method applied to VDM++ models is presented in chapter 3.

Mutation Testing

Mutation testing, which is a software testing strategy, is aimed to evaluate the effectiveness of test data used to test a model. It involves making subtle changes in the model and executing the original and the new models with the same test cases. The result of making the subtle changes in a model results in a new model which is called mutant. The subtle changes are called mutation operators. If the execution of the mutant and the original model with the same test case has a different output, then the test suite is considered effective at finding faults in the model [31]. On the other hand, if the execution of the mutant and the original model with the same test suite has the same output, then the mutant is equivalent to the original model. It is also possible that the model has redundancies. To understand if the model is redundant, a manual analysis should be carried throw by the user.

Although the advantage of evaluating the effectiveness of test cases, mutation testing is an expensive strategy due to the execution of a significant number of models for all test cases. Thus, a study about the model under test should be accomplished before applying this method in order to decide if the effort is worthwhile.

It is believed that in a considerable percentage of situations, this method could be helpful for testing in Formal Methods. When a model is tested with a large number of test cases and these have all a pass verdict, the tester should analyse whether the test cases were well chosen or not, covering the complete model under test. That is where mutation testing can help. Therefore, possible theoretical approaches of this method applied to VDM++ are explored in 3. Due to limitations of time, mutation testing will not be implemented in this thesis.

Unit Testing

A unit can be defined as the smallest possible testable software component [13]. Ideally, the units should be independent of each other. The main goal of unit testing is to be sure that each unit is implemented and working respecting its specification. Each VDM++ class can be considered a unit [9]. A framework of this testing strategy is described in [9].

1.2 The Aim Of This Thesis

The famous quote from Dijkstra [23] states ”Program testing can be used to show the presence of bugs, but never to show their absence!”. Thus, a possible definition for testing is that it is a way of gaining confidence in the model of a system under test [19]. However, this can be a repetitive, tedious , time consuming and expensive task, if there is no automation support, because the process

of testing is normally made with the purpose of covering the model as much as possible.

The aim of this thesis is to study possible test automation support for VDM++ and provide new mechanisms which can help testers to test VDM++ models in a more efficient way than with the current mechanisms available. Thus, besides a theoretical study about test automation for VDM++, a tool should also be developed so that some of the theory can be used in practice. The theoretical analysis should be clear enough so that someone who might be interested can start developing it. The tool is intended for all the audience who might want to test VDM++ models, independently of the size of the model.

Since this thesis is incorporated in an open source project called Overture, it is also a goal to enrich this project with test automation support.

1.3 Expected Outcomes

It is expected to achieve a number of outcomes from this thesis:

- Test automation in VDM++ is presented and it should enrich beginners in test automation, so that they can understand and apply the concept after reading this thesis.
- A combinatorial testing tool is expected to be specified in VDM++ and code generated in Java and from the resulted sources, an Eclipse Plugin should be built as part of the Overture Project. The tool should be built in a way that VDM++ users should be able to use it easily, dedicating almost no effort in understanding how the tool works. Thus, documentation about the tool should be clear for VDM++ users.
- Theoretical approaches of test automation in VDM++ should be described in order to enrich the available documentation about this subject. If a described theoretical approach can be applied in practice, an explanation about how could that be done should be provided so that someone can continue the research performed.

1.4 Outline of Thesis

An introduction of this thesis is provided in chapter 1. The testing tools that were analysed are presented in chapter 2. Theoretical approaches of the Mutation Testing strategy are described in chapter 3.

A tool was developed along this thesis and a detailed description about the tool is provided in chapter 4. Also a detailed description of the specification of the tool is provided in chapter 5. Finally, the concluding remarks are described in chapter 6.

Chapter 2

Testing Tools

This chapter provides a description of two test automation tools which were analysed during this thesis. VDM++TesK applies the Domain Testing and test sequencing strategies. This was the first tool being analysed. Unfortunately it was not possible to take advantage from this tool because although all the efforts that were made, the tool was not working properly. The support from the developers was not as efficient as it was asked and thus a deadend was reached before the goals for this thesis were achieved, when using VDM++TesK.

Tobias was the second tool being analysed and it applies the combinatorial testing strategy. It was possible to use the tool successfully, which helped to make the decision of developing a test automation tool for the Overture applying the combinatorial testing strategy. Although it would be possible to expand Tobias tool to VDM++, the sources were not available. Thus, the tool developed in this thesis was developed from scratch.

Another testing tool was supposed to be analysed, were mutation testing was applied [2]. However, the tool was not available and was not provided by the developers.

A tool framework where unit testing is applied is described in [9]. However, the automated work when using the tool is low. Thus, this tool is not described because this thesis is about test automation support and not only test support.

2.1 VDM++TesK: Testing of VDM++ programs

2.1.1 Introduction

VDM++TesK is a technology for test development for programs written in VDM++ [1], based on the UniTesK test development methodology [11].
A developer of VDM++ models can automate parts of test suite (A.3) design

and development, using VDM++TesK. This allows increasing the quality of the models, the reliability and other quality features of the target software.
With VDM++Tesk, the user is able to make oracle tests, based on a specification. This tool was built to be used with large and complex programs, or to make a large number of tests. The user will save time doing the tests using this tool and will probably make tests that one could not remember.
Testing small programs or making few tests will be worthwhile to test the software by hand. This tool is probably not so useful in these situations because the user will have to make an effort understanding how to use the tool and building the hand-made classes. In sub-section 2.1.2, there is a description of the hand-made classes.

2.1.2 General overview

VDM++TesK follows the *Design-by-Contract* concept. The main idea of this concept is that there is a contract between a class and its clients. Before calling a method from the class, the client must guarantee some conditions. In return, the class guarantees some properties that will remain after the call [15]. In this technology, the software contract is given in the form of pre-, postconditions and invariants of classes/ types.
The VDM++TesK test suite architecture was implemented as a collection of classes where each class has a specific functionality and it can be:

- Hand-made: the user must develop these classes by himself.
- Pre-built: these classes are already built.
- Generated: these classes can be automatically generated.

The complete VDM++TesK Test Suite Class Diagram can be found in figure 2.1.2.

To use this tool, the user should start building the specification respecting a few rules referred in [25]. With the specification, VDM++TesK is able to make partition analysis. This means that the test inputs will be divided in partitions, where each partition has inputs with similar behaviours. To know to which equivalence class a state belongs, it is used a test state identifier. The test state identifier is built basing on the current state of the system. In the test sequence iterator there is a component named ”coverage tracker” responsible for optimization: if a test was already done, it will not be tested again and the iterator will proceed to the next state. If the test was not done, it will be memorized in the tracker, tested and then the iterator proceeds to the next state. When all states are reached, the tests are finished.
Another problem besides the input data generation is getting the VDM model into valid states enabling certain tests. In other words, test sequences must be

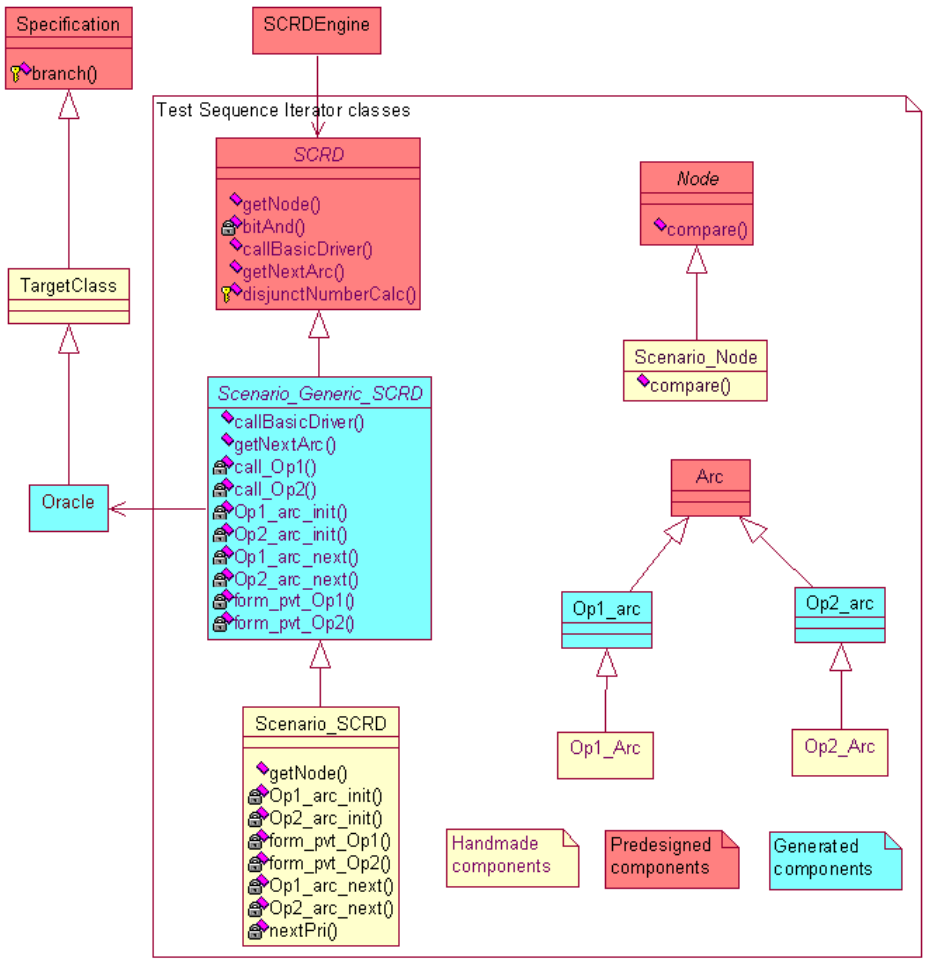

Figure 2.1: VDM++TesK Test Suite Class Diagram [25]

generated. VDM++TesK generates test sequences based on the test coverage completeness, which takes advantage of the post-conditions. According to the documentation, there are many test completeness criteria in VDM++TesK: the coverage of all post-condition branches (the weakest one), the coverage of all reachable combinations of logical terms that determine predicates of the corresponded branches and some more based on the post-conditions sub-branches. Normally, it is extremely hard for the machine to compute the arbitrary coverage criteria based on the specification or implementation due to the machine and time limitations. For this reason, VDM++TesK requires the user to underline the basic partition of the specified operation domain by means of branch constructors in the post-conditions of the specification.

The user has the possibility to test a particular sequence of operations, but one will have to change the generated code, in the operation ”call”. This operation executes the corresponding transition in the automaton under traversal.
The main goal of VDM++TesK is to have oracle tests. These are normally built automatically based on the specification and taking advantage of some operations from the test sequence iterator.
This section is only a general overview of VDM++TesK. For a more detailed description of the tool, including all the hand-made classes, [25] should be studied.

2.1.3 Critical analysis

In this section, it can be found a comparison between what the documentation says about VDM++TesK and what is it possible to do in practice with the tool.
This critical analysis is based on version 1.3 given by one of the developers of the tool. Probably there are more versions of VDM++TesK where this critical analysis makes no sense.
First of all, it was extremely hard to install the tool because there was no clear documentation about how to do it and the first version which was given did not have all the necessary files. After some communication with the developers, the tool was finally installed.
In version 1.3, there is an example called PQueue which works in VDM++TesK. Unfortunately, nothing was happening when one tried to make a new project besides a popup asking for the name of the project, i.e., it was not possible to insert the specification and implementation. After analysing and changing the Java code of the tool, it was possible to insert the referred files above using the GUI, but the tool was not able to generate any code. Thus, it was not possible to use this tool with any example except with the one which was already given by the developers.
It is hard to see in practice what it is possible to do with this tool having only one example and not being possible to generate another one from scratch. Therefore this critical analysis cannot be more detailed.

2.2 Tobias

2.2.1 Introduction

TOBIAS is a combinatorial testing tool which is aimed at the automatic production of test data (A.1), based on the name of methods and their inputs. The test data can then be transformed in test cases (A.2) adding the oracle with another tool (for this thesis, only VDMTools are used).

The model under test must have the specification written in JML [10] or VDM and the methods or functions must be written in Java [28] or VDM, respectively.

This tool can help the user to automatically generate test cases applying the combinatorial testing strategy (see sub-section 1.1.4) in lesser time than by hand and probably in a more elaborate way.

2.2.2 General overview

TOBIAS is a user friendly tool based on combinatorial testing. To generate test data, this tool transforms the input given by the user in regular expressions. Section 2.2.3 below explains how the regular expressions are used in combinatorial testing.
To use this tool, the user will have to select the input parameter values for given operations and in which order the operations should be called. With this information, the tool will generate in a chosen extension file (for example *.java* or *.arg*) all the combinations of the test data requested. This aims to assist the user to detect errors in the model under test.
In general, an *arg* file can be used with the VDMTools, associated with a VDM-SL or VDM++ file [8]. The *arg* file generated by Tobias can only be associated with a VDM-SL file. To associate with a VDM++ file, the *arg* file would have to be generated with a different syntax, where the function or operation would be applied to an object that has been created.
Since this thesis is about VDM++ Test Automation, the Java extension file will not be considered any further. Only the *arg* extension file will be developed even knowing that the developers of Tobias did not generate the *arg* file to be used with VDM++ specifications; it was generated to deal with VDM-SL specifications.

2.2.3 Detailed overview

This section provides a detailed description about Tobias and how the regular expressions are used in order to generate test data.
As written above, the user has to select the input parameter values and the order for the operations to be called. One can group specific methods with specific inputs and ask the tool to generate all possible permutations with repetitions. A small example is used below to take advantage of the capabilities described above.
The input below represents the call of three methods with specific inputs.

$$\begin{cases} Test = A(1)\ ;\ Aux\,\hat{}\{1\,..\,2\} \\ with\ Aux = \{B(x) \mid x \in \{1,\ldots,10\}\} \cup \{C(y) \mid y \in \{1,\ldots,5\}\} \end{cases}$$

The user is asking the tool to generate all possible tests with:

- The sequence of method A with input 1 only once and

- The union of method B with input from the set $\{1,...,10\}$ and method C with input from the set $\{1,...,5\}$ called once or twice (the 1..2 part).

The tool will generate all the possible combinations of those methods following the requested order. Note that asking for one to 2 repetitions with two methods means to ask for all possible permutations with one or two elements, with repetitions.
The permutations can be represented by the following regular expression:

$$AB \ + \ AC \ + \ ABB \ + \ ABC \ + \ ACB \ + \ ACC$$

which is equivalent to

$$AB \ (B \ + \ C \ + \ \epsilon) \ + \ AC \ (B \ + \ C \ + \ \epsilon)$$

However these are simply the possible permutations of the method invocations. The different arguments must also be considered.
In this small example there is a large number of test cases (15 + 15 x 15 = 240). In a realistic world example, the number of tests will be even larger and that will probably force the tool to ask for too much time and resources from the machine to run the tests and analyze the code. Thus, according to the documentation, TOBIAS filters test cases at *execution* and at *generation* time.
To filter test cases at *execution time*, TOBIAS has implemented a method which verdicts the result of each test case as INCONCLUSIVE, PASS or FAIL [29]. The result of the verdict depends if the implementation is respecting the specification.
When the input parameters does not respect the pre-condition, typing invariants or state invariant, then the verdict of the test case is INCONCLUSIVE.
If the behaviour of the operation conforms with the specification for the given input values and initial state, then the test has a PASS verdict.
If the operation or the output does not respect the typing invariants or state invariant, or the output does not respect the post-condition, then the test has a FAIL verdict.
Filtering at execution time does not require more input from the user. The tool takes advantage of the executable JML or VDM predicates.
The main principle of filtering at execution time is that if the beginning of a sequence of operations with a specific input has an INCONCLUSIVE or FAIL verdict then the next test case with the same prefix (both operation name and arguments) will not be tested because it would have the same verdict.
Filtering test cases at *generation time* requires the user to add VDM constraints to the executable model/code. The constraints are expected to pass in a VDM interpreter; the tests that fail the constraints are removed from the generated test suite.

Even with these filtering options, the user must be careful to manage combinatorial explosion issues; one should carefully design the test schemas.
Using VDMTools, it is possible to test a VDM model applying the test data generated by Tobias without having to spend a lot of time. VDMTools allow the user to add the oracle to the test data, taking advantage of the specification [22]. Thus, the user will be able to test ones model.
The generated test data cannot be applied in VDM++ models because the operations or functions are not applied to an instance of a class. Even so, this tool is useful for this thesis because its main idea, which is to generate combinatorial testing, can be applied for VDM++ Test Automation. However, it would be necessary to generate the test data with an adjusted syntax that allows the use of the principles of the object-oriented paradigm associated with the VDM++ specification language.

2.2.4 Limitations with respect to VDM++

This section provides a description about the advantages and disadvantages of supporting the principles of Tobias in a potential future VDM++ Test Automation context.
There are three main advantages in supporting the principles of Tobias in this thesis:

- The theory behind Tobias is clear and it is possible to apply it in VDM++ Test Automation;
- The level of automation is sufficiently high so it can help testing a VDM++ model in a more efficient way than by hand;
- It is not necessary to spend a long time understanding how to use the tool, as a user.

There is only one main disadvantage with Tobias as it stands right now and that is that it is not applying the principles of the object-oriented paradigm. Thus, if one creates an object of a given class using its constructor, it is not possible to call one or more methods over that object with Tobias. The methods will have to be changed in order to receive the parameters instead of being applied to objects.
This conclusion can be made from the grammar that the *twr* file must respect. It is not possible to declare an object. The grammar can be found below and it was copied from the documentation of Tobias [16].

tp	$::=$	**begin project**$(ClassDecl)^+$ **end project**
$ClassDecl$	$::=$	**begin class** $MyIDENTIFIER\ (MethodsDecl)^*$ **end class**
$MethodDecl$	$::=$	**method** $MyIDENTIFIER\ Parameters$
$Parameters$	$::=$	$(\ (\ Parameter(\ ,\ Parameter\ )^*\ )?\ )$
$Parameter$	$::=$	$TypeDef\ (\ ArrayOpt)?\ MyIDENTIFIER\ (ArrayOpt)?$
$TypeDef$	$::=$	$MyIDENTIFIER\ (\ \mathbf{.}\ MyIDENTIFIER\ )^*$
$ArrayOpt$	$::=$	**[ ]**
$MyIDENTIFIER$	$::=$	$< IDENTIFIER >$

with <IDENTIFIER> defined as follows:
<IDENTIFIER : ([**a**-**z**,**A**-**Z**,_,**$**] ([**a**-**z**,**A**-**Z**, _,$,**0**-**9**])*) >

So it is not impossible to construct test of the object-oriented paradigm with this grammar. However, there is no theoretical limitation in extending the testing strategy supplied by Tobias for this thesis.

2.2.5 Case study

In this section, two case studies are provided: the VDM-SL specification of the Triangle Problem [21] and the Towers of Hanoi [26].
The objective of the Triangle Problem is to classify a triangle as *isosceles*, *equilateral* or *scalene*, having as input three natural numbers corresponding to the size of its three sides. If the input is invalid in order to form a triangle, then it is not respecting the pre-condition and the user will be informed about that.
The tested function of this case study can be found below. The complete specification of the Triangle Problem can be found in sub-section B.1.1.

```
functions

classifyTriangleNats :
  nat *
  nat *
  nat ->
  triangleType
classifyTriangleNats(sideA, sideB, sideC) ==
  if (sideA = sideB) and (sideB = sideC)
  then mk_triangleType(<EQUILATERAL>)
  else (
    if (sideA <> sideB) and (sideB <> sideC) and (sideA <> sideC)
    then mk_triangleType(<SCALENE>)
    else mk_triangleType(<ISOSCELES>))
pre sideA > 0 and sideB > 0 and sideC > 0 and
       let perim = sideA + sideB + sideC
       in (2*sideA) < perim and (2*sideB) < perim and (2*sideC) < perim
post true;
```

In order to test this function with Tobias, it is necessary to write a *twr* file which respects a grammar that can be found in section 2.2.4. From this file, a *tob* file will be generated, using the Tobwritter, which is a Java program described in Tobias documentation [16]. The *tob* file is the input for the Tobias tool.
Thus, it is possible to generate the test data in a few minutes. The function classifyTriangleNats was tested with the following regular expression:

$$classifyTriangleNats(\{1,2,3\},\{2,4,9\},\{1,2,5\})$$

Tobias will generate all possible combinations with those inputs (3 * 3 * 3 = 27 test cases).
The beginning and end of the file can be found below. The complete file is in appendix C.1.1.

```
"Cas de test 1",
classifyTriangleNats(1,2,1),
"Cas de test 2",
classifyTriangleNats(2,2,1),
"Cas de test 3",
classifyTriangleNats(3,2,1),
...
"Cas de test 25",
classifyTriangleNats(1,9,5),
"Cas de test 26",
classifyTriangleNats(2,9,5),
"Cas de test 27",
classifyTriangleNats(3,9,5)
```

As it can be seen in the code, Tobias did not eliminate the cases which do not respect the specification. That happened because it was not possible to have filtering at execution time working, as well as filtering at generation time.
Using VDMTools, the specification is tested with the generated test data and the specification itself. The results are all coherent with the specification, as it can be seen in sub-section C.1.2.
Also using VDMTools, it was possible to conclude that a hundred percent line test coverage A.5 was reached for the method in cause, as it can be seen below.

Name	#Calls	Coverage
classifyTriangleNats	27	$\surd$
Total Coverage		**100%**

Figure 2.2: Test coverage for the Triangle Problem

Writing the generated test data by hand would certainly take longer than using the tool. It can be imagined that if the number of test cases increase to

hundreds or thousands, the difference between using the tool or generate the test cases by hand will be even more pronounced. It is not possible to generate

test sequences with the Triangle Problem because there is no state variable. For this reason, another example was made: the Towers of Hanoi with four disks [26]. The rules of the game are considered in the pre-condition of the specification. The goal of the game is not relevant to show the advantages of Tobias so when the goal is reached, the fact is ignored. Thus, if the sequence of moves respects the rules of the game, the test case will have a pass verdict.
For this example, three main operations were implemented being all similar. With each operation, it is possible to move a disk from a given peg to the peg with the number of the operation. These three operations were tested using Tobias and they can be found below. The complete model is in appendix B.1.2.

```
operations

-- move to peg1 the disk from peg x
move1 :
  nat ==>
  ()
move1(x) ==
  ( peg1 := peg1 ^ [top(x)];
    if (x=2)
    then peg2 := [peg2(a) | a in set inds peg2 & a < len(peg2)]
    else peg3 := [peg3(a) | a in set inds peg3 & a < len(peg3)]
  )
pre ((x=2 and peg2 <> []) or
     (x=3 and peg3 <> [])) and
    peg1 = [] or top(x) < top(1)
post peg1 = peg1~ ^ [top(1)] and
     ((x = 2) => peg2 ^ [top(1)] = peg2~) and
     ((x = 3) => peg3 ^ [top(1)] = peg3~);

-- move to peg2 the disk from peg x
move2 :
  nat ==>
  ()
move2(x) ==
  ( peg2 := peg2 ^ [top(x)];
    if (x=1)
    then peg1 := [peg1(a) | a in set inds peg1 & a < len(peg1)]
    else peg3 := [peg3(a) | a in set inds peg3 & a < len(peg3)]
  )
pre ((x=1 and peg1 <> []) or
     (x=3 and peg3 <> [])) and
     peg2 = [] or top(x) < top(2)
post peg2 = peg2~ ^ [top(2)] and
     ((x = 1) => peg1 ^ [top(2)] = peg1~) and
     ((x = 3) => peg3 ^ [top(2)] = peg3~);
```

```
-- move to peg3 the disk from peg x
move3 :
  nat ==>
  ()
move3(x) ==
  ( peg3 := peg3 ^ [top(x)];
    if (x=1)
    then peg1 := [peg1(a) | a in set inds peg1 & a < len(peg1)]
    else peg2 := [peg2(a) | a in set inds peg2 & a < len(peg2)]
  )
pre ((x=1 and peg1 <> []) or
     (x=2 and peg2 <> [])) and
      peg3 = [] or top(x) < top(3)
post peg3 = peg3~ ^ [top(3)] and
     ((x = 1) => peg1 ^ [top(3)] = peg1~) and
     ((x = 2) => peg2 ^ [top(3)] = peg2~);
```

The strategy to use this example in Tobias is similar to the one presented in the example above (Triangle Problem).
The above operations were tested with the following regular expression:

$$move3\ move2\ move1\ +\ move3\ move2\ move2\ move1$$

where each operation is called with a natural number from the set $\{1,2,3,4\}$.
Due to the large number of generated test cases, the complete output file from Tobias will not be shown in this thesis. The beginning and end of the file can be found below.

```
"Cas de test 1",
move3(1),
move2(1),
move1(1),
"Cas de test 2",
move3(1),
move2(1),
move1(2),
"Cas de test 3",
move3(1),
move2(1),
move1(3),
...,
"Cas de test 318",
move3(4),
move2(4),
move2(4),
```

```
move1(2),
"Cas de test 319",
move3(4),
move2(4),
move2(4),
move1(3),
"Cas de test 320",
move3(4),
move2(4),
move2(4),
move1(4)
```

Using VDMTools, the specification is tested with the generated test data and the specification itself. The results are all coherent with the specification, as it can be seen in sub-section C.1.2.
Using VDMTools, it was possible to conclude that a hundred percent line test coverage was reached for the operations in cause, as it can be seen below. Thus, the confidence of this model increased after using Tobias. There is no doubt that it would take longer to write by hand the test data generated by Tobias.

Name	#Calls	Coverage
move1	320	$\surd$
move2	576	$\surd$
move3	320	$\surd$
Total Coverage		**100%**

Figure 2.3: Test coverage for the Towers of Hanoi

Chapter 3

Mutation Testing

In this chapter, two theoretical approaches of the Mutation Testing strategy are presented, based on previous researches which are referred along the two sections.

3.1 Mutation Operators for VDM++

All through this sub-section, a comprehensive set of mutation operators for VDM++, which can be considered in a future development of a mutation testing tool, will be presented.
The quality of the mutation operators is crucial to the effectiveness of this method. They define the kind of faults that are detected in the mutant.
This theoretical approach is inspired in previous researches which are referred along the text below.

Sets of mutation operators can be divided in three levels which were suggested by Harrold and Rothermel [20], and Gallagher and Offutt [17]:

- Intra-method Level Faults: in this level, an analysis of the implementation of methods is made.
- Inter-method and Intra-class Level Faults: the interactions between pairs of methods of a single class is analysed in this level.
- Inter-class Level Faults: the object-oriented specific features are analysed in this level.

A description of each suggested mutation operator is given below as well as an example of each one.

Concerning the Intra-method Level Faults, the following mutation operators are presented:

- Expression Negation Operator: a logical expression is transformed in its negation.

Original	Mutant
expr(a) == if ***a*>*1*** then return 1 else return 0;	expr(a) == if ***not(a*>*1)*** then return 1 else return 0;

Table 3.1: Expression Negation Operator

- Binary Operator Replacement: the binary operators should be divided into classes, based on the input types which should be respected by the operator. For example, the mathematical binary operators can represent a class. A found mathematical binary operator is changed to another one from this same class, except the opposite, because that would generate the same mutant as the expression negation operator.

Original	Mutant
expr(a) == if ***a*>*1*** then return 1 else return 0;	expr(a) == if ***a*<*1*** then return 1 else return 0;

Table 3.2: Binary Operator Replacement

- Unary Operator Replacement: the unary operators should also be divided into classes, like the binary operators referred above. The strategy for these operators is similar with the one described above, about the binary operators.

Original	Mutant
expr(a) == if ***-a*>*1*** then return 1 else return 0;	expr(a) == if ***a*>*1*** then return 1 else return 0;

Table 3.3: Unary Operator Replacement

- Missing Condition Operator: An operand of a logical expression is removed.

Original	Mutant
expr(a,b) == if ***a*>*1* *or* *b*>*2+2*a*** then return 1 else return 0;	expr(a,b) == if ***b*>*2+2*a*** then return 1 else return 0;

Table 3.4: Missing Condition Operator

- Associative Shift Operator: the associativity of an expression is changed.

Original	Mutant
expr(a,b) == if b>***2+2*a*** then return 1 else return 0;	expr(a,b) == if b>***(2+2)*a*** then return 1 else return 0;

Table 3.5: Associative Shift Operator

Mutations in the Inter-method and Intra-class Level could be implemented by changing the order in which the methods are called. The probability of generating a mutant which is equivalent to the original method is considerably high, if this generation is made randomly. Another possibility would be to make sure that the output of a method is used in the next method, but verifying this would be extremely expensive. Therefore, this level is not considered relevant for mutation in VDM++, in this thesis.

The Inter-class Level Faults will be divided into three groups, based on the language features:

1. Information Hiding (Access Control)
2. Inheritance
3. Polymorphism

The described operators were proposed by Ma et al in [30]. Their choice was based on their experience as teachers in Object-Oriented software development and consulting with companies that also rely Object-Oriented software. Their research was made for Java language; therefore, an adaptation to VDM++ is proposed below.

1. Information Hiding

   VDM++ provides three access levels: public, private and protected. The incorrect use of these levels could lead to a possible fault. Therefore, a mutant operator modifying the access level is proposed. The main goal of this operator is to suggest the generation of test cases which ensure that accessibility is correct. It is possible that some of these mutants are equivalent to the original model because killing them or not depend on the features of the model.

Original	Mutant
public height : real;	***private*** height : real;

Table 3.6: Information hiding

2. Inheritance

   Object-oriented programming allows classes to inherit commonly used state and behaviour from other classes. This is a powerful and useful abstraction mechanism, but faults can occur if it is not used correctly. Thus, some mutation operators are proposed in order to try to cover the main possible faults based on inheritance.
   For the examples below, four classes were built: Bicycle, MountainBike, which is a subclass of Bicycle, Brand and Model, which is a subclass of Brand. Both Bicycle and MountainBike have the instantiated variable height.

   - Hiding variable deletion: an overriding declaration is deleted. In the mutant the use of the deleted variable in the subclass refers to the parent variable.

Original	Mutant

class Bicycle;
 	
instance variables;
 	
public wheels : nat := 2;
public height : real;
public brand : Brand := new Brand();
...
end Bicycle
 	
class MountainBike is subclass of Bicycle
 	
instance variables
public gears : nat;
public height : real;
	class Bicycle;
	
	instance variables;
	
	public wheels : nat := 2;
	public height : real;
	public brand : Brand := new Brand();
	...
	end Bicycle
	
	class MountainBike is subclass of Bicycle
	
	instance variables
	public gears : nat;
	- - public height : real;

Table 3.7: Hiding variable deletion

   - Hiding variable insertion: an overriding declaration is inserted. In the mutant, the use of the inserted variable in the subclass refers to the new variable instead of the parent variable.
   - Overriding method deletion: an overriding method is deleted. In the mutant, the use of the deleted method in the subclass refers to the parent method.
     It is expensive to verify if a method is defined both in the super class and in the subclass. Therefore, a possible different approach to apply this mutation could be to delete a method in the subclass without

Original	Mutant

class Bicycle;
 	
instance variables;
 	
public wheels : nat := 2;
public height : real;
public brand : Brand := new Brand();
...
end Bicycle
 	
class MountainBike is subclass of Bicycle
 	
instance variables
public gears : nat;
	class Bicycle;
	
	instance variables;
	
	public wheels : nat := 2;
	public height : real;
	public brand : Brand := new Brand();
	...
	end Bicycle
	
	class MountainBike is subclass of Bicycle
	
	instance variables
	public gears : nat;
	public height : real;

Table 3.8: Hiding variable insertion

verifying if it is an overriding method. An error will occur if it is not an overriding method and consequently the new model is not considered a mutant.

Original	Mutant

class Bicycle;
 	
instance variables;
 	
public wheels : nat := 2;
public height : real;
public brand : Brand := new Brand();
...
end Bicycle
 	
class MountainBike is subclass of Bicycle
...
operations
...
public heightVal : () ==> real
heightVal() == return height;
	class Bicycle;
	
	instance variables;
	
	public wheels : nat := 2;
	public height : real;
	public brand : Brand := new Brand();
	...
	end Bicycle
	
	class MountainBike is subclass of Bicycle
	...
	operations
	...
	- - public heightVal : () ==> real
	- - heightVal() == return height;

Table 3.9: Overriding method deletion

- Super class reference deletion: a method or a variable of a super class which is overriding can be used by the subclass if the name of

the super class is followed by an apostrophe and finally by the name of the variable or method. If the super class reference is deleted and the variable or method is defined in the subclass, then the variable or method after the change is the one from the subclass instead of the super class.

Original	Mutant
class Bicycle; instance variables; public height : real; ... end Bicycle class MountainBike is subclass of Bicycle ... operations ... public heightVal : () ==> real heightVal() == return ***Bicycle'height;***	class Bicycle; instance variables; public height : real; ... end Bicycle class MountainBike is subclass of Bicycle ... operations ... public heightVal : () ==> real heightVal() == return ***height;***

Table 3.10: Super class reference deletion

3. Polymorphism

   In a super class, the object references can refer to objects whose actual types are not the same as their declared types. The actual types can be any subclass or the class itself. Therefore, some polymorphism mutation operators are suggested.
   For the examples presented below, it is important to recall that "Brand" is a class independent from Bicycle and MountainBike and "Model" is a subclass of Brand. Both Brand and Model have the instantiated variable "name".

   - Change instantiated type: this operator changes the instantiated type of an object reference. Thus, the declaration of an object is maintained but the initialization is changed to a different subclass or the super class itself.
   - Change declaration type: this operator is the opposite from the change instantiated type operator. Thus, an object declaration of a method is changed to a subclass or the super class itself, but the initialization is maintained.

Original	Mutant
class Bicycle; instance variables; ... public brand : Brand := new ***Brand();*** ...	class Bicycle; instance variables; ... public brand : Brand := new ***Model();*** ...

Table 3.11: Change instantiated type

Original	Mutant
class Bicycle; instance variables; ... public brand : ***Brand*** := new Brand(); ...	class Bicycle; instance variables; ... public brand : ***Model*** := new Brand(); ...

Table 3.12: Change declaration type

- Change parameter variable declaration: this operator is similar with the change declaration type operator. The difference is that with this operator the changes are made on methods parameters instead of instance variables. Thus, it changes the declared type of a parameter object reference to a different subclass or to the super class itself.

Original	Mutant
class Bicycle; ... public getBrand : ***Brand*** ==> seq of char getBrand(b) == return b.name; ...	class Bicycle; ... public getBrand : ***Model*** ==> seq of char getBrand(b) == return b.name; ...

Table 3.13: Change parameter variable declaration

- Delete empty constructor: if a class has no empty constructors, VDM++ will create one by default. Therefore, it is suggested to delete the built empty constructor, if it exists. The main goal of this operator is to check if the empty constructor defined by the user is implemented properly.

Original	Mutant
class Bicycle ... ***public Bicycle : () ==> Bicycle*** ***Bicycle() == (height := 0.5);*** ...	class Bicycle ... ***- - public Bicycle : () ==> Bicycle*** ***- - Bicycle() == (height := 0.5);*** ...

Table 3.14: Delete empty constructor

3.2 Mutation and test case generation algorithm

The previous approach could be extended by adding another approach suggested by Bernhard Aichernig, from where a prototype tool is already implemented [2].

In [3], a test case generation algorithm is suggested, based on mutate the specifications in order to model errors that can occur during the development process.

The algorithm consists on finding a test case t(i,O), where i is the input and O is a set of outputs, which respects one of the conditions below. It is given a specification with pre and postcondition D(Pre $\vdash$ Post) and its mutant D'(Pre' $\vdash$ Post').

First condition: Pre $\wedge$ Post' $\wedge \neg$ Post

If it is possible to find a pair (i,o) which respects this condition, then test case t(i,O) is generated by finding a maximal solution (i,O) of

Pre $\wedge$ Post' $\wedge$ (v = ic)

If it is not possible to find a pair (i,o) which respects the condition, then the algorithm is followed by looking for a test case T = t(i,O) with (i,O) being a maximal solution of the second condition.

Second condition: $\neg$ Pre' $\wedge$ Pre $\wedge$ Post

If it is not possible to find a test case which satisfies one of the above conditions, then it is possible to conclude that the original and the mutant specifications are equivalent.

This algorithm is partial because the possible values of the variables in each test case are restricted to a finite domain, defined by the pre-condition.

A prototype tool of this algorithm is provided in [2].

Chapter 4

Combinatorial Testing tool

Along this chapter, a detailed description of the Combinatorial Testing tool can be found. Figure 4.1 is a schema representing the most important steps and structures of the Combinatorial Testing strategy specification and implementation. The figure is referred along with the description, so that the user can see how far one is from the final results, while reading the description.

Section 4.2 summarizes the purpose of Combinatorial Testing. A description of the concepts that should be understood in order to understand all the process of the combinatorial testing tool is followed in sections 4.3 and 4.4. Section 4.5 describes the development of the Combinatorial Testing tool. Finally, the limitations of the tool are described in section 4.6. A detailed description of the specification is provided in chapter 5.

4.1 Introduction

Combinatorial testing is the first test automation strategy considered for integration in the Overture platform. Such a strategy resorts to VDM++ models to generate the minimum number of test inputs in order to cover all the selected combinations of input values. Thus, test data is generated automatically from traces with chosen operations from a VDM++ model and chosen inputs selected by a tester. In the implementation of this strategy, it is important to automatically filter the irrelevant test data in order to reduce the possibility of having combinatorial explosion. However, the tester must also carefully design the model and the traces over it to take this risk into account.

The main interest of combinatorial testing is to automatically generate test sequences based on operations or functions from a model and chosen inputs. A specific algorithm will be used in order to filter the irrelevant test sequences.

The tool under development is inspired on another combinatorial testing tool,

called Tobias [22], which is aimed at the automatic production of test data, based on the name of functions and their inputs. The test data can then be transformed in test cases adding oracles with other tools (e.g. VDMTools [8]). A description about Tobias is provided in section 2.2.

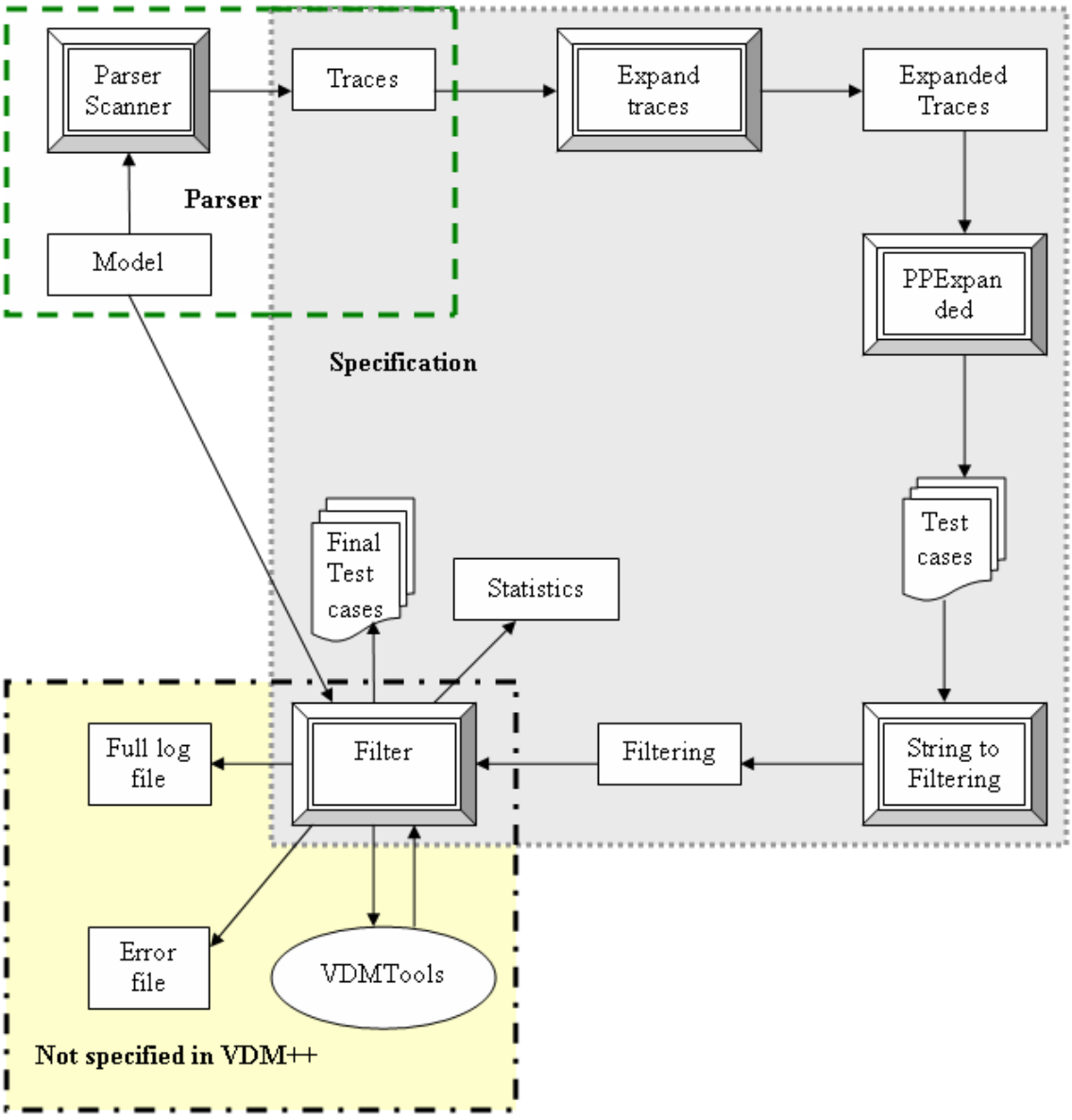

Figure 4.1: Combinatorial Testing schema

4.2 Purpose of Combinatorial Testing

When a VDM++ model is being tested with ordinary test cases, a tester will have to manually write the test cases one by one, and therefore one will have an exhaustive task.

The proposed Combinatorial Testing strategy can automate this task because the tester will have the possibility of writing traces representing the test scenarios one wishes to use. The test cases will then be automatically generated from the trace. They will also automatically be executed in the VDMTools interpreter and the output retrieved by the interpreter will be displayed.

The tester must be careful with the chosen traces due to the possibility of combinatorial explosion. A filtering process is performed, in order to remove the useless test cases. However the tester has a high responsibility to avoid combinatorial explosion, which is a disadvantage of this testing strategy.

4.3 Traces

As it is described in section 4.2, a tester has to write traces representing test cases in a compressed representation, in order to automatically generate the correspondent test cases.

VDM++ classes have some defined keywords which represent the name of different groups: instance variables, functions, operations, etc. The Overture Modelling Language (OML) supported by the Overture parser was extended with another group in order to allow the possibility of writing traces in a class, from where test cases will be automatically generated. The VDMTools parser will be updated with the same syntatic extensions as well. After the keyword *traces*, the user can write as many traces as one wants. The user must respect the following grammar, when defining the group *traces*:

```
traces definitions = 'traces', { named trace } ;

named trace = identifier , ':',trace definition list ;

trace definition list = trace definition term , { ';', trace definition term } ;

trace definition term = trace definition
                      | trace definition term , '|', trace definition ;
```

```
trace definition = trace core definition
                 | trace bindings , trace core definition
                 | trace core definition , trace repeat pattern
                 | trace bindings , trace core definition , trace repeat pattern ;

trace core definition = trace apply expression
                      | trace bracketed expression ;

trace apply expression = identifier , '.', Identifier , '(', expression list , ')' ;

trace repeat pattern = '*'
                     | '+'
                     | '?'
                     | '{', numeric literal , '}'
                     | '{', numeric literal , ',' numeric literal , '}' ;

trace bracketed expression = '(', trace definition list , ')' ;

trace bindings = { trace binding } ;

trace binding = 'let', list of local definitions , 'in'
              | 'let', bind , 'in'
              | 'let', bind , 'be', 'st', expression , 'in' ;
```

Due to the large number of lines required to present all the grammar for the *traces definition*, the definitions that are common with the existing VDM++ language are not defined here. However, they can be found in the VDM++ language manual [8].

A simple example where the traces block is respecting the grammar is:

```
traces

trace1 : s.Push3(1)*
trace2 : s.Push3(1)+
```

The traces can be written by using trace repeat patterns, as it is presented in the grammar. The meaning of each trace repeat pattern is:

- **a*** : repeat expression **a** from 0 to a pre-defined maximum value. If the maximum value is 3, then the retrieved value for this trace would be {nil, a, aa, aaa}.
- **a+** : repeat expression **a** from 1 to a pre-defined maximum value. If the maximum value is 3, then the retrieved value for this trace would be {a, aa, aaa}.
- **a?** : repeat expression **a** from 0 to 1. The retrieved value for this trace would be {nil, a}.
- **a**{**m**} : repeat expression **a** exactly **n** times. If m is equal to 4, the retrieved value would be {aaaa}.
- **a**{**n,m**} : repeat expression **a** between **n** and **m** times. If n is equal to 2 and m is equal to 4, the retrieved value would be {aa, aaa, aaaa}.

Expression *a* has limitations which are described in section 4.6.

In the example below, a specification of a stack model, it is possible to see how traces are associated to a model so that this can be tested using the combinatorial testing tool. It is also possible to see examples of traces with the structures suggested above.

```
class Stack

instance variables
  stack : seq of nat := [];
  s : Stack := new Stack();

operations

public Push3 : nat ==> ()
Push3(e) ==
      stack := [e] ^ stack
pre len stack < 3
post stack = [e] ^ stack~;

public Pop : () ==> nat
Pop() ==
  def res = hd stack in
  (stack := tl stack;
  return res)
pre stack <> []
post stack~ = [RESULT]^stack;
```

```
traces

trace1 : s.Push3(1)*
trace2 : s.Push3(1)+
trace3 : s.Push3(1)?
trace4 : s.Push3(1){2}
trace5 : s.Push3(1){0,4}; s.Pop()

end Stack
```

From this class, the generated test cases from each trace would be:

- **trace1: s.Push3(1)***:
  nil
  s.Push3(1);
  s.Push3(1); s.Push3(1);
  s.Push3(1); s.Push3(1); s.Push3(1);
  ...
  The maximum number of repetitions is defined by a variable associated to this operator *.

- **trace2: s.Push3(1)+**:
  s.Push3(1);
  s.Push3(1); s.Push3(1);
  s.Push3(1); s.Push3(1); s.Push3(1);
  ...
  The maximum number of repetitions is defined by a variable associated to this operator +.

- **trace3: s.Push3(1)?**:
  nil
  s.Push3(1);

- **trace4: s.Push3(1)**{**2**}:
  s.Push3(1); s.Push3(1)

- **trace5: s.Push3(1)**{**0,4**}:
  s.Pop()
  s.Push3(1); s.Pop()
  s.Push3(1); s.Push3(1); s.Pop()
  s.Push3(1); s.Push3(1); s.Push3(1); s.Pop()
  s.Push3(1); s.Push3(1); s.Push3(1); s.Push3(1); s.Pop()

As it can be seen from the above example, the variables to where the methods are called are defined in the *instance variables* group. It is important to notice that at the end of each generated test case, the variables are initialized automatically so that each test case is independent from each other. However, in each line the variables are not initialized. Thus, trace5 called in the VDMTools interpreter would be equivalent to:

```
s.Pop()
init
s.Push3(1); s.Pop()
init
s.Push3(1); s.Push3(1); s.Pop()
init
s.Push3(1); s.Push3(1); s.Push3(1); s.Pop()
init
s.Push3(1); s.Push3(1); s.Push3(1); s.Push3(1); s.Pop()
init
```

The possibility of writing sequences of test cases can reduce the effort the tester would have to spend without this tool, specially when the tester needs to test a specific state of the model which requires a sequence of method calls.

The user can also define traces which will call a method with different input parameters. Using the same specification above, some possible traces could be:

```
...

traces

trace6 :  let x in set {1,5,10} in
          s.Push3(x); s.Pop()
trace7 :  let x in set {1,...,3} in
          s.Push3(x); s.Pop()
trace8 :  let x = 1 in
          s.Push3(x)
...
```

The test cases automatically generated for each trace would be:

- **trace6**:
  Push3(1); Pop()
  Push3(5); Pop()
  Push3(10); Pop()

- **trace7**:
  Push3(1); Pop()
  Push3(2); Pop()
  Push3(3); Pop()

- **trace8**:
  Push3(1)

Finally, besides a sequence of method calls, the user can also use the choice symbol, represented by a pipe | in a trace definition term. Thus, considering the following trace:

```
traces

trace9 : s.Push1(2) | s.Push1(3)
```

The following test cases would be:

- s.Push1(2)

- s.Push1(3)

Limitations of using the expression let and let-be-such-that in the developed tool are described in section 4.6.

In this section, it was explained how traces are transformed into test cases. Traces are the structures that represents the input given by the tester. The test cases are a compressed representation of an ordered sequence of invocations of specific operations with parameters. Therefore, it is necessary to expand the Traces structure so that the test cases can be executed by the VDMTools interpreter.

4.4 Filtering process - theoretical approach

In order to be able to filter test cases, each test case, made of sequences of object creations or method calls, with or without parameters, will have an INCONCLUSIVE, PASS or FAIL verdict. The result of the verdict depends on whether the implementation verifies the specification.
Whenever the input parameters violate the pre-condition, type invariants or the state invariant, the verdict of the test case is INCONCLUSIVE.
If the operation conforms to the specification for the given input values and initial state, then the test has a PASS verdict.
If the operation or the output does not respect the type invariants or the state

invariant, or the output violates the post-condition, the test produces a FAIL verdict.

In other words, if an INCONCLUSIVE verdict is reached, the chosen test case did not respect the specification and thus it is the fault of the testers traces description that the error occur. However, if a FAIL verdict is reached it can be concluded that the specification is not correct and thus the specifier should review and correct it.

The idea underlying such a filtering process is that, if a sequence of operations for a specific input has an INCONCLUSIVE or FAIL verdict, then any following test cases with the same prefix does not need to be tested as it will lead to the same verdict.

In order to see an example where these concepts are applied, the stack model specified in the section above will be considered, as well trace5:

```
...
trace5 : s.Push3(1){0,4}; s.Pop()
```

The trace5 would generate the following test cases:

```
s.Pop()\\
s.Push3(1); s.Pop()\\
s.Push3(1); s.Push3(1); s.Pop()\\
s.Push3(1); s.Push3(1); s.Push3(1); s.Pop()\\
s.Push3(1); s.Push3(1); s.Push3(1); s.Push3(1); s.Pop()
```

The first test case would have an INCONCLUSIVE verdict because the pre-condition of the operation *Pop* is not satisfied, and also an INCONCLUSIVE verdict would be reached in the last test case because the pre-condition of the operation *Push3* is not satisfied. All the other test cases of this trace would have a PASS verdict.

In order to have a FAIL verdict, the specification would have to be changed. Thus, the following specification is considered:

```
class Stack

instance variables
  stack : seq of nat := [];
  inv len stack < 3;
  s : Stack := new Stack();

operations

public Push3 : nat ==> ()
```

```
Push3(e) ==
      stack := [e] ^ stack
post stack = [e] ^ stack~;

traces

trace10 : s.Push3(1); s.Push3(1); s.Push3(1)

end Stack
```

The pre-condition of the operation Push3 was removed and an invariant was added to the instance variable stack. Thus, trace10 would have a FAIL verdict because the invariant is not respected.

4.5 Development of the Combinatorial Testing Tool

The tool under development is itself almost entirely modelled in VDM++. From this model, Java code is generated automatically. A few aspects of the tool will be hand implemented in Java. Finally, it would be possible to assemble the Java code into an Eclipse plugin.

In the sequel we provide a brief overview of the intended support tool. Its first stage resorts to a parser to collect user input according to a pre-defined grammar. The Overture Modelling Language and consequently the Overture parser were extended so that the user can write traces directly in a VDM++ model. Thus, a new class keyword called *traces* was created. In this group, each trace is identified by a keyword defined by the user. An example of a class taking advantage of the new parser is available in section 4.3. The user is able to call methods with chosen sequences of inputs and ask for the generation of permutations with repetitions of a sequence of methods and inputs. User input is automatically encoded into a trace which guides the testing process in subsequent phases.

In order to proceed with further transformations, the Traces structure is transformed into another structure, which represents an expanded version of the first one. This new structure is represented in Figure 4.1 as *Expanded Traces*. There is no direct interaction between this structure and the user because it is a complex structure for the user to read and understand. Thus, it is only an intermediate structure for the specification, where the test cases are already expanded and available in a non-user-friendly representation. This structure cannot be executed by the VDMTools interpreter because it is not a string, which is the accepted input by the interpreter. For this reason, the *Expanded Traces* structure is transformed into another one called *Test cases* in Figure 4.1, which has an internal state with sequences of strings, where each sequence represents a generated test case.

All the test cases the tester required in the trace are already transformed when the specification reaches the state *Test cases*, from figure 4.1, in the form of strings. These strings are respecting the required rules from the VDMTools interpreter. Thus, they can be directly executed in the interpreter and the test generation process could be finished. However there is a high probability that some of the test cases can automatically be detected as useless test cases. For this reason, the test generation process is not complete at this point.

The filtering process will be explained theoretically below and then it is how that is applied in the specification, which is represented as *String to Filtering* and *Filter* in Figure 4.1.

After generating the test cases in an automatic way, the tool applies a filter process in order to eliminate irrelevant test cases, similar with the one developed in Tobias [29]. Four components will be delivered by the tool:

- **Full log**: all executed test cases will be printed in this file, associated with its own output retrieved by the VDMTools interpreter. Thus, each executed test case from this file could had had a PASS, FAIL or INCONCLUSIVE verdict.
- **Error file**: all executed test cases with a FAIL verdict will be printed in this file, also associated with its own output retrieved by the VDMTools interpreter. This file is probably the most relevant for the users because only the test cases that fail due to the fact that the specification was not correct will be here.
- **Statistics**: the number of executed, failed and deleted test cases as well as the percentage of failed and deleted test cases will be printed in the output.
- **Arg files**: All generated argument files will be saved because the user might be interested in using some of them in the future.

At the moment, only the specification is finished and thus, none of the files are retrieved because it is necessary to implement these features in Java. However, all the relevant information to generate the referred outputs is being held in the Statistics component.

4.6 Tool limitations

Due to the limited time available for this project, the following limitations are present in the described combinatorial testing tool:

- Constraints which might appear in a let-be-such-that are accepted if they are a set range, a set enumeration or a seq enumeration. Besides be-

ing accepted, they are also saved in a type structure. However, they are ignored and thus the test cases are generated ignoring the constraint.

- In a let or let-be expression, the bind expressions are limited to: symbolic literal expression, sequence enumeration, set enumeration, map enumeration, set range expression, name and binary expression. Any other expressions are not supported by this tool.
- *Trace bracketed expression*, suggested in the grammar of section 4.3, is not accepted.

Chapter 5

Specification description

This chapter provides a description of the specification built for the combinatorial testing tool. The specification is divided in five classes: *Expanded*, *PP* (Pretty-Printing), *Filtering*, *ToolBox* and a final class called *CTesting* where the main operations are invoked. Figure 5.1 contains the referred classes and the relation between each other. Thus, the structure of the specification is represented in this figure. Note that an arrow with origin in *class A* and destiny in *class B* means that *class A* is being used within *class B*.

A description of each class is provided in each correspondent section. The types of the specification are described in section 5.1. A critical comment for all the specification is that, exluding a few exceptions, the operations defined in the specification should be functions because functions are more abstract than operations and there is no reason to specify operations instead of functions along this project.

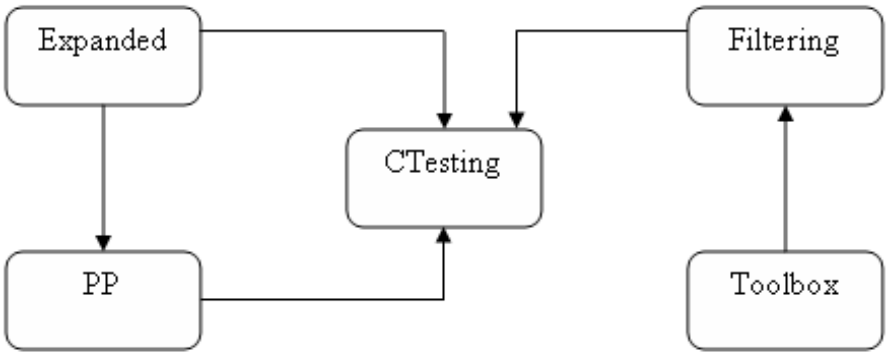

Figure 5.1: Structure of the specification

5.1 Types

The internal VDM++ types defined in the specification are based on the abstract syntax representation of the Overture Modelling Language (OML). In the Overture abstract syntax representation, the traces are available in a compressed representation. However it is necessary to transform the traces in an expanded representation.

The user is allowed to write more than one trace. Thus, each trace in the compressed representation corresponds to an expanded trace in the expanded representation, as it is specified below.

```
public TraceDefinitionsExpanded ::
  defs : seq of NamedTraceExpanded;

public NamedTraceExpanded ::
  name : seq of char
  defs : TraceDefinitionExpanded;
```

Each expanded trace must have an identifier, represented as *name* in the NamedTraceExpanded, and an expanded trace definition, represented as *defs*. Each expanded trace definition can be an item, a sequence of items or each item can be divided by the symbol choice. The TraceSequenceDefinitionExpanded and TraceChoiceDefinitionExpanded are sequences of TraceDefinitionExpanded and so they are equivalent to sequences of items. However, they are necessary in order to maintain the information available on the compressed representation of the trace.

```
public TraceDefinitionExpanded =
  TraceDefinitionItemExpanded |
  TraceSequenceDefinitionExpanded |
  TraceChoiceDefinitionExpanded;

public TraceSequenceDefinitionExpanded ::
  defs : seq of TraceDefinitionExpanded;

public TraceChoiceDefinitionExpanded ::
  defs : seq of TraceDefinitionExpanded;
```

Each expanded item is a sequence of method calls (which has a variable, a method name and a parameter list, e.g., var.method(parameter1, parameter2)) or bracketed expressions. Unfortunately, it was not possible to specify the acceptance of bracketed expressions due to the limited time provided.

```
public TraceDefinitionItemExpanded ::
  item_list : seq of TraceDefinitionItemElement;

public TraceDefinitionItemElement ::
  test    : TraceCoreDefinitionExpanded;

public TraceCoreDefinitionExpanded =
  TraceMethodApplyExpanded |
  TraceBracketedDefinitionExpanded;

public TraceMethodApplyExpanded ::
  methodList : seq of OmlTraceMethodApply;

public TraceBracketedDefinitionExpanded ::
  definition : TraceDefinitionExpanded;
```

As it can be seen in the grammar described in section 4.3, bindings can be written in a trace. A TraceBinding is defined in the Overture abstract representation as:

```
TraceBinding =
        TraceLetBinding |
        TraceLetBeBinding;

TraceLetBinding ::
  definition_list : seq of ValueShape;

TraceLetBeBinding ::
  bind : Bind
  best : [Expression];

ValueShape ::
  pattern    : Pattern
  type       : [Type]
  expression : Expression;
```

Thus, if a trace definition has a variable defined in a let or let-be expression, and it is used in a method, the expanded version of the method call (called *trace apply expression* in the parser) must replace the name of the variable by its value. For this reason, the values of the variables defined in a let or let-be expression are saved in a variable of the type TraceBindInformation specified below. The constraints that might be defined in a let-be-such-that are also saved in TraceBindInformation. However, they will not be considered when the expansion of the trace definition is made, as it is described in chapter 6.

```
public TraceBindInformation ::
  letIn : map seq of char to IOmlExpression
  letBe : map seq of char to IOmlExpression
  constraints : [seq of OmlExpression];
```

For a detailed description about the VDM++ types, the user manual [8] is referred.

5.2 CTesting

CTesting is the main class where the main operations are invoked. It receives the trace definitions and it retrieves the statistics, where the percentage and total of the deleted, selected and failed test cases will be present, as well as the selected and deleted test cases itself, in the current version of the specification. As future work, the test cases will be in the full log file and in the error file, as it is described in section 4.5. The constructor responsible for this task is provided below.

```
public CTesting :
  OmlTraceDefinitions *
  seq of (seq of char) ==>
  CTesting
CTesting(r, specs) ==
  (
  rex    := new Expanded();
  pp     := new PP();
  erex   := rex.expandTraces(r);
  pp_rex := pp.ppSeqDefItemExpanded(erex);
  filter := new Filtering(pp_rex, specs);
  stat   := filter.ppTestCases();
  );
```

After creating an object of the class Expanded and another of the class PP, the TraceDefinitions are expanded. The output must be pretty-printed and then filtered. Finally, the statistics and test cases are generated. A description of all this process is provided in the previous chapter. Below, a description of each class presented in figure 5.1 is provided.

5.3 Expanded

The class *Expanded* is the first step of the specification process because the traces are expanded here. The types of this class are based on the types defined in the Overture abstract syntax definition. It is necessary to map the con-

nection between these two structures, so the traces available in a compressed representation can be transformed in the expanded representation.

Along this section, the specification of this class will be explained through four different sub-sections, each one explaining one main collection of operators of this class. The sub-section 5.3.1 will consist in gathering the information from the let and let-be expressions. Furthermore, the sub-section 5.3.2 will explain in detail how a repetition pattern is expanded. The sub-section 5.3.3 will provide a description about the expansion of the core expressions based on the information gathered in the first collection of operators (let and let-be expressions). Finally, the sub-section 5.3.4 will explain how the combination of different elements of a trace is performed forming the final test cases.

In order to explain each section, only the main operations will be provided. The complete specification is provided in section D.1. Thus, auxiliary operations of the ones presented below should be seen in the referred section.

5.3.1 Let and let-be expressions

In this subsection it is explained how the variables of the let and let-be expressions are saved in the TraceBindInformation, so that the variables used in the core definition are changed by its value, as it is described in subsection 5.3.3. The main operation responsible for this task is defined as:

```
public getInformation :
  OmlTraceDefinitionItem ==>
  TraceBindInformation
getInformation(i) ==
  let b = i.getBind(),
      letBe = merge { getLetBeInfo(b(e)) |
         e in set inds b & isofclass(OmlTraceLetBeBinding,b(e)) },
      letIn = merge { getLetInfo(b(e)) |
         e in set inds b & isofclass(OmlTraceLetBinding,b(e)) },
      constraints = removeNil([ getConstraints(b(e)) |
         e in set inds b & isofclass(OmlTraceLetBeBinding,b(e)) ])
  in return mk_TraceBindInformation(letIn, letBe, constraints);
```

As it can be seen in operation *getInformation*, for each let or let-be expression found, another operation is applied in order to add the variable name and the expression associated and eventually the constraint to maps which will be sent as input parameters to the TraceBindInformation type. The auxiliary operation for the let expressions is provided below.

```
public getLetInfo :
  OmlTraceLetBinding ==>
  map seq of char to IOmlExpression
```

```
getLetInfo(b) ==
  let def_list = b.getDefinitionList()
  in return
      { getPatternId(def_list(e).getPattern())
        |->
        def_list(e).getExpression()
      | e in set inds def_list
        & isofclass(OmlPatternIdentifier,def_list(e).getPattern())
      };

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

An example where this operation would be required is:

```
trace11 : let x = 1 in var.method(x)
```

The variable x is added to the TraceBindInformation. Nothing more is done in this step. Thus, the variable x in the method call is not changed by its value, at this state.

5.3.2 Repetition pattern

This collection of operators is responsible for expanding expressions with symbols +, *, ? and {n,m}. It is important to notice that when a trace has these trace repeat patterns after a method call which has a variable whose value is defined in a let expression, the first step is to repeat the expression with the variable name, and the variable will only be changed by its possible values after this step.
The operation responsible for expanding traces with repeat patterns is defined as:

```
public expandRegexpr :
  OmlTraceDefinitionItem ==>
  seq of (seq of OmlTraceDefinitionItem)
expandRegexpr(i) ==
  return expandRegexprChoose(i);

public expandRegexprChoose :
  OmlTraceDefinitionItem ==>
  seq of (seq of OmlTraceDefinitionItem)
expandRegexprChoose(i) ==
  let r = i.getRegexpr()
```

```
  in if i.hasRegexpr()
     then return expandSymbol(i,r)
     else return [[i]];
```

The OmlTraceDefinitionItem can have different input types and thus a method call to decide which input is received is necessary. In VDM++ the next operation would not be necessary because it is possible to have different operations with the same name and different inputs, and when the operation is called, the type is inferred. However, when these operations are code generated to java, type inference will not be available. Thus, the next operation is necessary.

```
public expandSymbol :
  OmlTraceDefinitionItem  *
  IOmlTraceRepeatPattern ==>
  seq of (seq of OmlTraceDefinitionItem)
expandSymbol(s,r) ==
  cases true:
    (isofclass(OmlTraceZeroOrMore,r))
      -> return expandSymbolZeroOrMore(s,r),
    (isofclass(OmlTraceOneOrMore,r))
      -> return expandSymbolOneOrMore(s,r),
    (isofclass(OmlTraceZeroOrOne,r))
      -> return expandSymbolZeroOrOne(s,r),
    (isofclass(OmlTraceRange,r))
      -> return expandSymbolRange(s,r)
  end;
```

The OmlTraceDefinitionItem is then repeated as many times as it is required. All the auxiliary operations presented above, in operation *expandSymbol*, call the operation *expandN2M* described below with the appropriate input parameters. The output retrieved by *expandN2M* is built in the fourth input parameter recursively.

```
public expandN2M :
  nat *
  nat *
  OmlTraceDefinitionItem *
  seq of OmlTraceDefinitionItem ==>
  seq of (seq of OmlTraceDefinitionItem)
expandN2M(n,m,s,o) ==
  if n <> m
  then return  [o] ^ expandN2M(n,m-1,s, o ^ [s])
  else if n = 0
       then return [[new OmlTraceDefinitionItem([],
                     new OmlTraceMethodApply([],[],[]),nil)]]
       else return [o];
```

In order to explain this operation, it will be assumed that the third input parameter is a natural number instead of an OmlTraceDefinitionItem, so it will be more readable. After this change, an example of calling this operation is the following:

```
expandN2M(1,2,10,[10])
```

1 is different from 2 and so [10] is concatenated with expandN2M(1,1,a,[10,10]). In the recursion, n is equal to m (1=1). Thus, we have [10] concatenated with [10,10] which retrieves [[10],[10,10]]. In conclusion, 10 was repeated from once to twice after the execution of this operation. In analogy, if the input was an OmlTraceDefinitionItem instead of a natural number, the trace definition item would be repeated between once and twice. The following example can be considered:

```
trace12 : var.method(1){1,2}
```

This trace would be expanded by the operation *expandN2M* by the following, after transforming the retrieved output into strings, so that this can be readable:

```
var.method(1)
var.method(1); var.method(1)
```

5.3.3 Core expressions

This collection of operators is responsible for expanding the trace core definitions with variables. Thus, the variable names of the method calls are changed to its possible values. The same example considered in subsection 5.3.1 is also expanded by this collection of operators.

```
trace11 : let x = 1 in var.method(x)
```

At this state, it is possible to retrieve the following, after transforming the retrieved output into strings:

```
var.method(1)
```

It is only possible because the variable x was already added to the TraceBindInformation by the operation *getInformation*, explained in subsection 5.3.1. The operation responsible for this task is defined as:

```
public expandCoreDefinition :
  TraceBindInformation *
  seq of (seq of OmlTraceDefinitionItem) ==>
  TraceDefinitionItemExpanded
expandCoreDefinition(m,l) ==
  let e = conc [ expandSeqDefItem(m,l(i)) | i in set inds l ]
  in return mk_TraceDefinitionItemExpanded(e);
```

5.3.4 Combinations

This is the main collection of operators which retrieves the final expanded version of a trace. The operation responsible for this task is called *expandTraces* which is provided below.

```
public expandTraces :
  OmlTraceDefinitions ==>
  seq of TraceDefinitionItemExpanded
expandTraces(t) ==
  let trace = t.getTraces(),
      emptymap = mk_TraceBindInformation({|->},{|->},[]),
      sequence = conc [ combinationsSeq(expand(trace(e),emptymap))
                        | e in set inds trace]
  in return sequence;
```

In this operation, the TraceBindInformation is initialized and each trace is expanded by calling the operation *expand*. Thus, a sequence of expanded traces is retrieved.

In order to understand how the input combinations is made in the operation *combinationsSeq*, a polymorphic function was specified. The idea underlying the function is the same as in the mentioned operation, except that in the operation it is necessary to be careful with the possible types that might appear as input. The complete specification of the operation *combinationSeq* can be found in section D.1. An example where it is necessary to combine the input values is:

```
trace13 : let x in set {1,2} in var1.meth(x); var2.meth2(10)
```

The following test cases should be automatically generated:

```
var1.meth(1); var2.meth2(10)
var1.meth(2); var2.meth2(10)
```

The polymorphic function is provided and explained below. The input for this function with the previous example would be, after applying the value of x to each method call:

```
[[var1.meth(1), var1.meth(2)],[var2.meth(10)]]
```

```
functions

public combinations[@A] :
  seq of (seq of @A) ->
  seq of (seq of @A)
combinations(s) ==
  if s = [] then s
  else
  (
  let h = hd s,
      t = tl s,
      e = [[h(i)] | i in set inds h]
  in concAll[@A](e,t)
  );
```

The function combinations receives a sequence of sequences of elements of a given type and it will combine all elements present inside the first level of the sequence. Considering the first element of the first level of the sequence, it will be transformed in a sequence with itself as the only element.

```
public concAll[@A] :
  seq of (seq of @A) *
  seq of (seq of @A) ->
  seq of (seq of @A)
concAll(h,t) ==
  conc [ concFirst[@A]([h(i)],t)| i in set inds h];
```

As it can be seen in the function *concFirst*, the process of combinations is recursive and it is started by connecting the tail of the input with the first element of the head. This way, the test cases are being built while the sequence of elements is being consumed.

```
public concFirst[@A] :
  seq of (seq of @A) *
  seq of (seq of @A) ->
  seq of (seq of @A)
concFirst(t,s) ==
  (if s = [] then t
  else (
  let head = hd(s),
      tail = tl(s),
      mid  = concat[@A](t,head)
  in concFirst[@A](mid,tail)
```

```
      )
  );

public concat[@A] :
  seq of (seq of @A) *
  seq of @A ->
  seq of (seq of @A)
concat(relPaths,t) ==
  conc([ mapValue[@A](relPaths(i),t)
       | i in set inds relPaths]);
```

Finally, the function *mapValue* is the responsible for the concatenation of the elements of each test case.

```
public mapValue[@A] :
  seq of @A *
  seq of @A ->
  seq of seq of @A
mapValue(relPath,t) ==
  [ relPath ^ [t(i)] | i in set inds t ];
```

5.4 Pretty-Printing

This class was specified in order to transform the expanded test cases in strings, so that these can be readable and essentially to be executed in the VDMTools interpreter. The main operation of the Pretty-Printing is defined as:

```
public ppSeqDefItemExpanded :
  seq of Expanded‘TraceDefinitionItemExpanded ==>
  seq of (seq of Expanded‘BasicType)
ppSeqDefItemExpanded(d) ==
  return [ ppDefExpanded(d(i)) | i in set inds d];
```

It can be seen from the operation above that the input from this operation is the output from the operation *expandTraces* showed in section 5.3. It is responsible for pretty-printing those results to a sequence of sequences of *BasicType*. A *BasicType* is as follows:

```
public BasicType =
  char |
  bool |
  real |
  seq of BasicType |
  seq of (BasicType * BasicType);
```

This type was created in order to be hold the information contained in the Oml abstract syntax definition. Thus, each time there is a need of evaluate a Oml Expression, its value will be transformed into a BasicType.

5.5 Filtering

The Filtering process is applied in the specification starting by transforming the test cases into the domain of the map *allTestCases*, from the specification presented below. This map is created in a class which is represented in figure 4.1 as *Filtering*.

```
types

public Output = (nat * seq of char);

public TestCases ::
  allTestCases : map (seq of char) to Output
  failedTestCases : map (seq of char) to nat;

operations

public Filtering :
  seq of (seq of char) *
  seq of (seq of char) ==>
  Filtering
Filtering(t, specs) ==
  testCases.failedTestCases := {|->};
```

The range of *allTestCases*, called Output, is structured with a natural number and a string. The natural number represents if the associated test case was not tested, or was tested and had an associated verdict, and therefore the number can have three different values. The string has, if necessary, the output retrieved by the interpreter, after an element of the domain of *allTestCases* is executed. As it can be seen in the specification above, the domain of *allTestCases* is initialized with all the test cases and the range is initialized with 0, which means that the test case was still not analyzed, and an empty string, because it was not executed and therefore there is no output retrieved by the interpreter. The map *failedTestCases* from the specification above will have all the test cases which do not have a PASS verdict. It is also created in the class *Filtering*. When the filtering process is initialized, no test cases where executed and therefore it is not possible to define its verdict. Thus, the *failedTestCases* is initialized as an empty map. Each test case is executed in the VDMTools interpreter, as it is represented in figure 4.1, after the following conditions are satisfied:

1. The test case was not yet tested;

2. The prefix of the test case is not present in the domain of the map with the test cases that failed.

If the second condition is not satisfied then the test case will not be executed because the verdict would be INCONCLUSIVE or FAIL, as it is explained in the previous sub-section. Consequently, the test case will be inserted in the *failedTestCases* map and the associated range will be changed. The test case will also be removed from the *allTestCases* map. Thus, at the end of the filtering process, this map will correspond to the map with the executed test cases.
If both conditions are satisfied, then the test case is executed. After the execution, the range of the test case will be appropriately changed and the test case will be added to the *failedTestCases* map, if it had an INCONCLUSIVE or FAIL verdict. This part of the process is represented as *Filter* in figure 4.1.

It is possible to specify the process of calling the VDMTools interpreter, taking advantage of Dynamic Link Classes (DLClasses). However, DLClasses will not be used due to the simplicity of this process. Thus, the external call for the interpreter will be implemented in Java. This external call is still not implemented. An operation is specified in VDM++, so that the evaluation of the verdict for each test case is done in a fictitious way. Thus, all test cases are specified to fail, at the moment.

It is finally possible to retrieve the final results to the tester. The selected and failed test cases are printed separately (represented as *Final Test Cases*, in figure 4.1) and the tester is informed about the number of executed, failed and deleted test cases as well as the percentage of failed and deleted test cases (represented as *Statistics*, in figure 4.1). Two files will also be automatically generated: full log file and error file, as it is represented in figure 4.1. The full log file will contain all the executed test cases associated with its own output retrieved by the VDMTools interpreter. Thus, each executed test case from this file could had had a PASS, FAIL or INCONCLUSIVE verdict. The error file will have all executed test cases with a FAIL verdict, also associated with its own output retrieved by the VDMTools interpreter. This file is probably the most relevant for the users because only the test cases that fail due to the fact that the specification was not correct will be here.

As it can be seen in figure 4.1, the connection to the VDMTools, in the filtering process, is not specified in VDM++ due to its simplicity; it will be implemented in Java.

5.6 ToolBox

This class has a very simple specification because it will be extended in Java. In order to have the verdict of each test case, it is necessary to execute them in the VDMTools interpreter. Only the structure where the verdict and the output retrived by the interpreter will be saved is specified, as it can be seen below.

```
public interpreterResult ::
        verdict : nat
        output : seq of char
inv x == x.verdict in set {0,1,2};
```

Thus, the operation vdmToolsCall will be extended in Java. At the moment, all test cases are considered to have a FAIL verdict.

```
public vdmToolsCall :
  seq of char ==>
  interpreterResult
vdmToolsCall(-) ==
  return mk_interpreterResult(2,"Failed");
```

Chapter 6

Concluding Remarks

Along this chapter, an analysis of the results of this project is provided. In section 6.1, the achieved results are described and they are compared with the initial intensions of the project. In section 6.2, the hardest challenges that were reached along this thesis are described. Finally the future work is suggested in section 6.3.

6.1 Achieved Results

The specification of the combinatorial testing tool accepts almost all trace definitions which were intended to be considered. Thus, it will be possible to automate a considerable amount of work in the process of testing VDM++ models. After writing traces, a tester will automatically obtain all the requested test scenarios. The specification is ready to be automatically generated into Java. A few errors appear in the Java code due to small bugs of the java code generator of the VDMTools interpreter; however, they are easily solved. In order to add the filtering process, the specification is also ready with the necessary structures. Only Java is necessary to connect the specification with the VDMTools interpreter. Thus, a new mechanism is provided in order to help testing VDM++ models in a more efficient way.

In order to use the specification to automatically generate test cases, it is necessary to generate the chosen traces into an OML value using the Overture parser and then create a VDM++ file with the outcome value, followed by adding the specification to the VDMTools interpreter and call the operation *expandTraces* over the created value. It was initially planned that this process would be fully automatic. However, as it is discrebed in section 6.2, due to time limitations it was not possible to have a more user-friendly tool.

A theoretical approach of the Mutation Testing strategy applied to VDM++ is

is suggested in section 3.1. Thus, if one is interested in developing a Mutation Testing tool for test automation in VDM++, some theoretical work was already done and thus one can start by developing what it is suggested and extend it with additional possibilities. A detailed theoretical approach of the tool developed in this thesis is also provided.

In subsections 1.1.4 and 1.1.5, a survey of existing testing strategies which can be applied to VDM++ test automation have been provided. Thus, these subsections give background for those who might be interested in test automation for VDM++. References are made along these subsections and also along all the thesis, so that it is possible to know where one can find more information about what is being described and have a deeper study, if necessary.

VDM++ will be extended with the goal of accepting *traces definitions*, described in section 4.3. Due to the relevance of this construct, VDMTools will extend the VDM++ language with this functionality.

6.2 Discussion

Along this thesis, the following challenges were achieved:

- Without having any background in testing, learning testing without traditional lessons was the first challenge. Although this thesis is only about test automation for VDM++, it was necessary to have a larger background in order to understand the concepts. The most basic concepts are not provided in detail in documentation about test automations for Formal Methods and thus it was important to read documentation about testing in general. After some time, this challenge was overtaken.

- VDMTesK was the first testing tool being analysed and, although a large effort was spent in order to use this tool, success did not appear as it was initially envisaged, as it is described in section 2.1. Support from the developers of this tool was not as constructive as was hoped and it was not possible to take advantage of the VDMTesK tool and these principles.

- After deciding that a combinatorial testing tool should be developed and after start working on the parser side, the Overture community concluded that extending the Overture parser was impossible for newcommers. Thus, it was decided that the Overture parser should be extended to allow the usage of traces. It was then decided that the specification should be started, independently on the parser, and later the parser and the specification would be connected. When the parser was ready, the specification was ready and the implementation in Java was already started. However, the specification had to be made from scratch because it was not matching the parser. The initial idea of how the parser was going to be developed

had some changes in order to have a better work, which required a new specification to be made in a short period of time. It was planned that in this short period of time, more theoretical approaches of test automation for VDM++ would be described and an Eclipse plugin would be developed. However it was not possible because the time was needed for developing the specification.

6.3 Future Work

The most important future work item is to implement the connection between the specification and the VDMTools interpreter. An Eclipse plugin of the tool should also be developed. Furthermore, the limitations described in section 4.6 should be overtaken.

It would also be useful to turn this tool to be more independent from the user. Domain testing [13] could be specified in order to avoid the work that testers have to spend on choosing inputs. Even so, it is probably important to maintain the possibility of allowing the tester to choose inputs because it might be ones intention to test specific test cases, in some situations. A description of Domain Testing is provided in subsection 1.1.4.

Mutation testing [31] could also be specified in order to verify how well the chosen test cases had covered a model. Theoretical approaches of this strategy are also described in chapter 3.

Finally, the generated test cases should be saved in files, as it is described in section 4.5. Also the arg files which are generated to execute each test case should be saved in order to make regression testing possible.

Bibliography

[1] A.K. Petrenko A.A. Koptelov, V.V. Kuliamin. Vdm++ TesK: Testing of VDM++ programs. July 2002. [cited at p. 9]

[2] Bernhard K. Aichernig and Percy Antonio Pari Salas. Test case generation by ocl mutation and constraint solving. In *QSIC*, pages 64–71, 2005. [cited at p. 9, 28, 29]

[3] B.Aichernig. On the value of fault injection on the modeling level. JULY 2005. [cited at p. 28]

[4] H.Q.Nguyen C.Kaner, J.Falk. *Testing Computer Software*. Wiley, 1999. [cited at p. 4, 6]

[5] J.Bach C.Kaner. Paradigms of Black Box Software Testing, November 1998. Available at: http://www.kaner.com/pdfs/swparadigm.pdf. [cited at p. 5]

[6] S.Dalal C.Lott, A.Jain. Modeling Requirements for Combinatorial Software Testing. May 2005. [cited at p. 5]

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

[8] CSK Systems Corporation. VDMTools User Manual (VDM++) 1.2. Technical report, CSK, 2007. [cited at p. 13, 32, 34, 46]

[9] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs For Object-oriented Systems*. Springer-Verlag TELOS, Santa Clara, CA, USA, 2005. [cited at p. 2, 7, 9]

[10] Albert L. Baker Gary T. Leavens 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. 12]

[11] V.V. Kuliamin I.B. Bourdonov, A.S. Kossatchev and A.K. Petrenko. UniTesK Test Suite Architecture. 2002. [cited at p. 9]

[12] http://jflex.de/. [cited at p. 4]

[13] I.Burnstein J.Francois. *Pratical Software Testing*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002. [cited at p. 7, 59, 65]

[14] Cem Kaner. Black box software testing: A course by Cem Kaner & James Bach, 2005. Available at: http://www.testingeducation.org/BBST/Domain.html. [cited at p. 5]

[15] Gary T. Leavens and Yoonsik Cheon. Design by Contract with JML. Draft, available from jmlspecs.org., 2005. [cited at p. 10]

[16] Y. Ledru. Documentation for Tobwriter v0.1, July 2004. [cited at p. 15, 17]

[17] A.Offutt L.Gallagher, J.Offutt. Integration testing of object-oriented components using finite state machines. *Softw. Test. Verif. Reliab.*, 16(4):215–266, 2006. [cited at p. 21]

[18] 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. [cited at p. 3]

[19] Christophe Meudec. *Automatic Generation of Software Test Cases From Formal Specifications*. PhD thesis, Faculty of Science of the Queen's University of Belfast, May 1998. [cited at p. 7]

[20] G.Rothermel M.Harrold. Performing data flow testing on classes. In *SIGSOFT '94: Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, pages 154–163, New York, NY, USA, 1994. ACM. [cited at p. 21]

[21] N D North. Automatic Test Generation for the Triangle Problem. Technical report, National Physical Laboratory, February 1990. [cited at p. 16]

[22] P. Bontron O. Maury, Y. Ledru and L. Bousquet. Using TOBIAS for the automatic generation of VDM test cases. 2002. [cited at p. 15, 32]

[23] C. Hoare O.J. Dash, E. Dijkstra. *Structured Programming*, volume 8 of *APIC Studies in Data Processing*. Academic Press, 1972. [cited at p. 7]

[24] http://www.overturetool.org. [cited at p. 3]

[25] P.G.Larsen. *VDM++ Test Automation: VDMTesK*. CSK, 2007. [cited at p. 10, 11, 12, 107]

[26] wikipedia. Tower of Hanoi. [cited at p. 16, 18]

[27] Laurie Williams. Testing Overview and Black-Box Testing Techniques, 2004. Available at: http://agile.csc.ncsu.edu/SEMaterials/BlackBox.pdf. [cited at p. 6]

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

[29] O. Maury Y. Ledru, L. Bousquet and P. Bondron. Filtering TOBIAS Combinatorial Test Suites. 2004. [cited at p. 14, 41]

[30] Y.Kwon Y.Ma, J.Offutt. Inter-class mutation operators for java. In *Software Reliability Engineering, 2002. ISSRE 2002. Proceedings. 13th International Symposium on*, pages 352–363, 2002. [cited at p. 23]

[31] Y.Kwon Y.Ma, J.Offutt. Mujava : An automated class mutation system. *Softw. Test. Verif. Reliab.*, 15(2):97–133, June 2005. [cited at p. 7, 59]

Appendices

Appendix A

Testing Terminology

Along this chapter, a description of the basic definitions about Testing which are used in the thesis is provided.

A.1 Test data

Test data is a set of test inputs given by a hardware, software or human, without any oracle.

A.2 Test case

A test case is an item with, at least, the following information [13]:

- A set of test inputs: data items given by a hardware, software or human.
- Execution conditions: it explains what are the required conditions to run the test.
- Expected outputs: which result should the tester except.

The test case can have more information in order to increase its value as a reusable object, or simply to provide more useful information to a tester or developer.
There are many other definitions for Test Case. The difference is that they can have more or less information than the definition above. The main idea of test case is always to gain information about the program.

A.3 Test suite

A test suite is a set of test cases possibly organized in a hierarchical structure.

A.4 Test oracle

A test oracle assigns a test case with a pass or fail verdict.
In the context of this thesis, the oracle is based on the relation between the implementation and the specification. In other words, one test case has a pass verdict if the implementation for the test case respects the specification. Otherwise it has a fail or inconclusive verdict assigned by the test oracle.

A.5 Test coverage

The test coverage is the percentage of the model that has been tested. There are many variants of test coverage. In this thesis, line test coverage is referred and the meaning is that all the lines of a model are tested at least once.

Appendix B

VDM Models of the Case Studies

B.1 Tobias Models

B.1.1 triangle.vdm

```
values

tri : triangle = mk_triangle(1,1,1);

types

triangle :: sideA : nat
            sideB : nat
            sideC : nat
inv t ==
  t.sideA > 0 and t.sideB > 0 and t.sideC > 0
  and
  let perim = t.sideA + t.sideB + t.sideC
  in (2*t.sideA) < perim and (2*t.sideB) < perim and
     (2*t.sideC) < perim;

triangleType :: <EQUILATERAL> | <ISOSCELES> | <SCALENE>;

functions

classifyTriangleNats :
  nat *
  nat *
  nat ->
  triangleType
classifyTriangleNats(sideA, sideB, sideC) ==
  if (sideA = sideB) and (sideB = sideC)
  then mk_triangleType(<EQUILATERAL>)
  else (
```

```
    if (sideA <> sideB) and (sideB <> sideC) and (sideA <> sideC)
    then mk_triangleType(<SCALENE>)
    else mk_triangleType(<ISOSCELES>))
pre sideA > 0 and sideB > 0 and sideC > 0 and
  let perim = sideA + sideB + sideC
  in (2*sideA) < perim and (2*sideB) < perim and (2*sideC) < perim
post true;
```

B.1.2 hanoi.vdm

```
types

state tower of
    peg1 : seq of nat -- disks in peg1 with size 1, 2, 3 or/and 4
    peg2 : seq of nat -- disks in peg2 with size 1, 2, 3 or/and 4
    peg3 : seq of nat -- disks in peg3 with size 1, 2, 3 or/and 4
inv mk_tower(a,b,c) ==
      len a <=4 and len b <= 4 and len c <= 4
      init t == t = mk_tower([4,3,2,1],[],[])
end

operations


-- It returns the disk which is on the top of peg p
top :
  nat ==>
  nat  -- number of peg
top(p) ==
  if (p=1) then return peg1(len(peg1))
           else if (p=2) then return peg2(len(peg2))
                else return peg3(len(peg3))
pre p<=3 and (if p=1
              then peg1 <> []
              else if p=2 then peg2 <> []
                          else peg3 <> []);


 -- move to peg1 the disk from peg x
move1 :
  nat ==>
  ()
move1(x) ==
  ( peg1 := peg1 ^ [top(x)];
  if (x=2) then peg2 := [peg2(a) | a in set inds peg2 & a < len(peg2)]
           else peg3 := [peg3(a) | a in set inds peg3 & a < len(peg3)]
  )
pre ((x=2 and peg2 <> []) or
     (x=3 and peg3 <> [])) and
    peg1 = [] or top(x) < top(1)
```

```
post peg1 = peg1~ ^ [top(1)] and
     ((x = 2) => peg2 ^ [top(1)] = peg2~) and
     ((x = 3) => peg3 ^ [top(1)] = peg3~);

-- move to peg2 the disk from peg x
move2 :
  nat ==>
  ()
move2(x) ==
  ( peg2 := peg2 ^ [top(x)];
    if (x=1)
    then peg1 := [peg1(a) | a in set inds peg1 & a < len(peg1)]
    else peg3 := [peg3(a) | a in set inds peg3 & a < len(peg3)]
  )
pre ((x=1 and peg1 <> []) or
     (x=3 and peg3 <> [])) and
     peg2 = [] or top(x) < top(2)
post peg2 = peg2~ ^ [top(2)] and
     ((x = 1) => peg1 ^ [top(2)] = peg1~) and
     ((x = 3) => peg3 ^ [top(2)] = peg3~);

-- move to peg3 the disk from peg x
move3 :
  nat ==>
  ()
move3(x) ==
  ( peg3 := peg3 ^ [top(x)];
    if (x=1)
    then peg1 := [peg1(a) | a in set inds peg1 & a < len(peg1)]
    else peg2 := [peg2(a) | a in set inds peg2 & a < len(peg2)]
  )
pre ((x=1 and peg1 <> []) or
     (x=2 and peg2 <> [])) and
      peg3 = [] or top(x) < top(3)
post peg3 = peg3~ ^ [top(3)] and
     ((x = 1) => peg1 ^ [top(3)] = peg1~) and
     ((x = 2) => peg2 ^ [top(3)] = peg2~);
```

Appendix C

Generated output from studied tools

C.1 Tobias output

C.1.1 triangle.arg

```
"Cas de test 1",
classifyTriangleNats(1,2,1),
"Cas de test 2",
classifyTriangleNats(2,2,1),
"Cas de test 3",
classifyTriangleNats(3,2,1),
"Cas de test 4",
classifyTriangleNats(1,4,1),
"Cas de test 5",
classifyTriangleNats(2,4,1),
"Cas de test 6",
classifyTriangleNats(3,4,1),
"Cas de test 7",
classifyTriangleNats(1,9,1),
"Cas de test 8",
classifyTriangleNats(2,9,1),
"Cas de test 9",
classifyTriangleNats(3,9,1),
"Cas de test 10",
classifyTriangleNats(1,2,2),
"Cas de test 11",
```

```
classifyTriangleNats(2,2,2),
"Cas de test 12",
classifyTriangleNats(3,2,2),
"Cas de test 13",
classifyTriangleNats(1,4,2),
"Cas de test 14",
classifyTriangleNats(2,4,2),
"Cas de test 15",
classifyTriangleNats(3,4,2),
"Cas de test 16",
classifyTriangleNats(1,9,2),
"Cas de test 17",
classifyTriangleNats(2,9,2),
"Cas de test 18",
classifyTriangleNats(3,9,2),
"Cas de test 19",
classifyTriangleNats(1,2,5),
"Cas de test 20",
classifyTriangleNats(2,2,5),
"Cas de test 21",
classifyTriangleNats(3,2,5),
"Cas de test 22",
classifyTriangleNats(1,4,5),
"Cas de test 23",
classifyTriangleNats(2,4,5),
"Cas de test 24",
classifyTriangleNats(3,4,5),
"Cas de test 25",
classifyTriangleNats(1,9,5),
"Cas de test 26",
classifyTriangleNats(2,9,5),
"Cas de test 27",
classifyTriangleNats(3,9,5)
```

C.1.2 VDMTools Output for the Triangle Problem

```
Parsing "../Tobias/Tobias1.6.1/TOBIAS/Triangle/triangle.vdm"
 (Latex) ... done
Initializing specification ... done
"Cas de test 1"
../Tobias/Tobias1.6.1/TOBIAS/Triangle/triangle.vdm, l. 82, c. 44:
  Run-Time Error 58: The pre-condition evaluated to false
```

```
"Cas de test 2"
mk_DefaultMod‘triangleType( <ISOSCELES> )
"Cas de test 3"
../Tobias/Tobias1.6.1/TOBIAS/Triangle/triangle.vdm, l. 82, c. 44:
  Run-Time Error 58: The pre-condition evaluated to false
"Cas de test 4"
../Tobias/Tobias1.6.1/TOBIAS/Triangle/triangle.vdm, l. 82, c. 44:
  Run-Time Error 58: The pre-condition evaluated to false
"Cas de test 5"
../Tobias/Tobias1.6.1/TOBIAS/Triangle/triangle.vdm, l. 82, c. 44:
  Run-Time Error 58: The pre-condition evaluated to false
"Cas de test 6"
../Tobias/Tobias1.6.1/TOBIAS/Triangle/triangle.vdm, l. 82, c. 44:
  Run-Time Error 58: The pre-condition evaluated to false
"Cas de test 7"
../Tobias/Tobias1.6.1/TOBIAS/Triangle/triangle.vdm, l. 82, c. 44:
  Run-Time Error 58: The pre-condition evaluated to false
"Cas de test 8"
../Tobias/Tobias1.6.1/TOBIAS/Triangle/triangle.vdm, l. 82, c. 44:
  Run-Time Error 58: The pre-condition evaluated to false
"Cas de test 9"
../Tobias/Tobias1.6.1/TOBIAS/Triangle/triangle.vdm, l. 82, c. 44:
  Run-Time Error 58: The pre-condition evaluated to false
"Cas de test 10"
mk_DefaultMod‘triangleType( <ISOSCELES> )
"Cas de test 11"
mk_DefaultMod‘triangleType( <EQUILATERAL> )
"Cas de test 12"
mk_DefaultMod‘triangleType( <ISOSCELES> )
"Cas de test 13"
../Tobias/Tobias1.6.1/TOBIAS/Triangle/triangle.vdm, l. 82, c. 44:
  Run-Time Error 58: The pre-condition evaluated to false
"Cas de test 14"
../Tobias/Tobias1.6.1/TOBIAS/Triangle/triangle.vdm, l. 82, c. 44:
  Run-Time Error 58: The pre-condition evaluated to false
"Cas de test 15"
mk_DefaultMod‘triangleType( <SCALENE> )
"Cas de test 16"
../Tobias/Tobias1.6.1/TOBIAS/Triangle/triangle.vdm, l. 82, c. 44:
  Run-Time Error 58: The pre-condition evaluated to false
"Cas de test 17"
```

```
../Tobias/Tobias1.6.1/TOBIAS/Triangle/triangle.vdm, l. 82, c. 44:
  Run-Time Error 58: The pre-condition evaluated to false
"Cas de test 18"
../Tobias/Tobias1.6.1/TOBIAS/Triangle/triangle.vdm, l. 82, c. 44:
  Run-Time Error 58: The pre-condition evaluated to false
"Cas de test 19"
../Tobias/Tobias1.6.1/TOBIAS/Triangle/triangle.vdm, l. 82, c. 44:
  Run-Time Error 58: The pre-condition evaluated to false
"Cas de test 20"
../Tobias/Tobias1.6.1/TOBIAS/Triangle/triangle.vdm, l. 82, c. 44:
  Run-Time Error 58: The pre-condition evaluated to false
"Cas de test 21"
../Tobias/Tobias1.6.1/TOBIAS/Triangle/triangle.vdm, l. 82, c. 44:
  Run-Time Error 58: The pre-condition evaluated to false
"Cas de test 22"
../Tobias/Tobias1.6.1/TOBIAS/Triangle/triangle.vdm, l. 82, c. 44:
  Run-Time Error 58: The pre-condition evaluated to false
"Cas de test 23"
mk_DefaultMod‘triangleType( <SCALENE> )
"Cas de test 24"
mk_DefaultMod‘triangleType( <SCALENE> )
"Cas de test 25"
../Tobias/Tobias1.6.1/TOBIAS/Triangle/triangle.vdm, l. 82, c. 44:
  Run-Time Error 58: The pre-condition evaluated to false
"Cas de test 26"
../Tobias/Tobias1.6.1/TOBIAS/Triangle/triangle.vdm, l. 82, c. 44:
  Run-Time Error 58: The pre-condition evaluated to false
"Cas de test 27"
../Tobias/Tobias1.6.1/TOBIAS/Triangle/triangle.vdm, l. 82, c. 44:
  Run-Time Error 58: The pre-condition evaluated to false
```

Appendix D

Combinatorial Testing Specification

D.1 Expanded.vpp

Below, one can find the specification of the class Expanded. First, the types used to hold the expanded informationof the traces are shown.

```
class Expanded

types

-------------------------------------------------------
-- TraceDefinitions from the Overture AST are
-- redefined here as expanded
-------------------------------------------------------
public TraceDefinitionsExpanded ::
  defs : seq of NamedTraceExpanded;

public NamedTraceExpanded ::
  name : seq of char
  defs : TraceDefinitionExpanded;

public TraceDefinitionExpanded =
  TraceDefinitionItemExpanded |
  TraceSequenceDefinitionExpanded |
  TraceChoiceDefinitionExpanded;

public TraceSequenceDefinitionExpanded ::
  defs : seq of TraceDefinitionExpanded;

public TraceChoiceDefinitionExpanded ::
```

```
  defs : seq of TraceDefinitionExpanded;

public TraceDefinitionItemExpanded ::
  item_list : seq of TraceDefinitionItemElement;

public TraceDefinitionItemElement ::
  test    : TraceCoreDefinitionExpanded;

public TraceCoreDefinitionExpanded =
  TraceMethodApplyExpanded |
  TraceBracketedDefinitionExpanded;

public TraceMethodApplyExpanded ::
  methodList : seq of OmlTraceMethodApply;

public TraceBracketedDefinitionExpanded ::
  definition : TraceDefinitionExpanded;
```

In order to holdthe information within the let expressions, the type *TraceBindInformation* was criated. Furthermore, the BasicType is presented, which will hold the result of evaluate an expression such as a sequence enumeration.

```
-----------------------------------------------------
-- Values of the variables defined in letIn and
-- letBe expressions are saved in this structure
-- for further treatment
-----------------------------------------------------

public TraceBindInformation ::
  letIn : map seq of char to IOmlExpression
  letBe : map seq of char to IOmlExpression
  constraints : [seq of OmlExpression];

public BasicType =
  char |
  bool |
  real |
  seq of BasicType |
  seq of (BasicType * BasicType);
```

The instance variables responsible for holding the limit value being used by regular expressions are shown below.

```
instance variables

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

```
-- Maximum number of repetitions of the operators
-- + and * are defined here
-----------------------------------------------------

private zeroOrMoreMax : nat := 3;
private oneOrMoreMax : nat := 3;
```

The main operation of this specification is named *expandTraces*, and its presented below.

```
operations

-----------------------------------------------------
-- Main operation which expand a sequence of traces --
-----------------------------------------------------


public expandTraces :
  OmlTraceDefinitions ==>
  seq of TraceDefinitionItemExpanded
expandTraces(t) ==
  let trace = t.getTraces(),
      emptymap = mk_TraceBindInformation({|->},{|->},[]),
      sequence = conc [ let l  = combinationsSeq(expand(trace(e),
                                                 emptymap)),
                            op = getMainOperator(trace(e))
                        in if op = true then l else normalize(l)
                        | e in set inds trace]
  in return sequence;
```

Some useful operations in order to normalize the result of the main operation.

```
public getMainOperator :
  OmlNamedTrace ==>
  bool
getMainOperator(t) ==
  let defs = t.getDefs()
  in return getValueFromOperator(defs);


public getValueFromOperator :
  IOmlTraceDefinition ==>
  bool
getValueFromOperator(t) ==
```

```
  cases true:
    (isofclass(OmlTraceDefinitionItem,t))
      -> return false,
    (isofclass(OmlTraceSequenceDefinition,t))
      -> return true,
    (isofclass(OmlTraceChoiceDefinition,t))
      -> return true,
    others
      -> return false
  end;


public normalize :
  seq of TraceDefinitionItemExpanded ==>
  seq of TraceDefinitionItemExpanded
normalize(l) ==
  return conc [ normalizeElements(l(i)) | i in set inds l];


public normalizeElements :
  TraceDefinitionItemExpanded ==>
  seq of TraceDefinitionItemExpanded
normalizeElements(t) ==
  let defs = t.item_list
  in if len defs = 1
     then return [t]
     else return normalization(defs);


public normalization :
  seq of TraceDefinitionItemElement ==>
  seq of TraceDefinitionItemExpanded
normalization(s) ==
  return [ mk_TraceDefinitionItemExpanded([s(i)]) | i in set inds s];
```

The operation below is the responsible for expanding one trace definition. The remain operations are helper operations in order to achieve the necessary results.

```
-----------------------------------------------------
-- Expand One Trace Definition ----------------------
-----------------------------------------------------

public expand :
  OmlNamedTrace *
  TraceBindInformation ==>
  NamedTraceExpanded
expand(t,m) ==
  let name = t.getName(),
```

```
      defs = t.getDefs(),
      newDefs = expandDefs(defs,m)
  in return mk_NamedTraceExpanded(name,newDefs);

-----------------------------------------------------
-- Expand Oml Definition ----------------------------
-----------------------------------------------------

public expandDefs :
  IOmlTraceDefinition *
  TraceBindInformation ==>
  TraceDefinitionExpanded
expandDefs(defs,m) ==
  cases true:
    (isofclass(OmlTraceDefinitionItem,defs))
      -> return expandItem(defs,m),
    (isofclass(OmlTraceSequenceDefinition,defs))
      -> return expandSequence(defs,m),
    (isofclass(OmlTraceChoiceDefinition,defs))
      -> return expandChoice(defs,m)
  end;
```

The operation that expands a sequence definition is presented below.

```
-----------------------------------------------------
-- Expand a TraceSequenceDefinition -----------------
-----------------------------------------------------


public expandSequence :
  OmlTraceSequenceDefinition *
  TraceBindInformation ==>
  TraceSequenceDefinitionExpanded
expandSequence(s,m) ==
  let d = s.getDefs(),
      l = conc [ expandSequenceHelper(d(i),m) | i in set inds d ]
  in return mk_TraceSequenceDefinitionExpanded(l);


public expandSequenceHelper :
  IOmlTraceDefinition *
  TraceBindInformation ==>
  seq of TraceDefinitionExpanded
expandSequenceHelper(e,m)==
  cases true:
    (isofclass(OmlTraceDefinitionItem,e))
      -> return [expandItem(e,m)],
    (isofclass(OmlTraceSequenceDefinition,e))
      -> return [expandSequence(e,m)],
```

```
    (isofclass(OmlTraceChoiceDefinition,e))
      -> return [expandChoice(e,m)],
    others
      -> return []
  end;
```

Furthermore, the operation *expandChoice* is responsible for expanding the choice definition.

```
-----------------------------------------------------
-- Expand TraceChoiceDefinition ---------------------
-----------------------------------------------------

public expandChoice :
  OmlTraceChoiceDefinition *
  TraceBindInformation ==>
  TraceChoiceDefinitionExpanded
expandChoice(c,m) ==
  let d = c.getDefs(),
      l = conc [ expandChoiceHelper(d(i),m) |  i in set inds d]
  in return mk_TraceChoiceDefinitionExpanded(l);

public expandChoiceHelper :
  IOmlTraceDefinition *
  TraceBindInformation ==>
  seq of TraceDefinitionExpanded
expandChoiceHelper(e,m) ==
  cases true:
    (isofclass(OmlTraceDefinitionItem,e))
      -> return [expandItem(e,m)],
    (isofclass(OmlTraceSequenceDefinition,e))
      -> return  [expandSequence(e,m)],
    (isofclass(OmlTraceChoiceDefinition,e))
      -> return [expandChoice(e,m)],
    others
      -> return []
  end;
```

Concerning the item, it is expanded using the operation below.

```

-----------------------------------------------------
-- Expand Item --------------------------------------
-----------------------------------------------------

public expandItem :
  OmlTraceDefinitionItem *
  TraceBindInformation ==>
```

```
  TraceDefinitionItemExpanded
expandItem(i,n) ==
  let m = getInformation(i),
      u = traceBindInfoUnion(m,n),
      tc = expandRegexpr(i),
      s = expandCoreDefinition(u,tc)
  in  return s;
```

Whenever the core of a trace needs to be expanded, the operation *expandCoreDefinition* is used. This is used when there is a need of expand the parameters of a given method, by consulting the information present in the let expressions.

```
----------------------------------------------------------------------
-- Expand Core Definition ----------------------------------------
----------------------------------------------------------------------

--Responsible for expanding methods that have variables in the
--let/let-be bind.
public expandCoreDefinition :
  TraceBindInformation *
  seq of (seq of OmlTraceDefinitionItem) ==>
  TraceDefinitionItemExpanded
expandCoreDefinition(m,l) ==
  let e = conc [ expandSeqDefItem(m,l(i)) | i in set inds l ]
  in return mk_TraceDefinitionItemExpanded(e);

public expandSeqDefItem :
  TraceBindInformation *
  seq of OmlTraceDefinitionItem ==>
  seq of TraceDefinitionItemElement
expandSeqDefItem(m,s) ==
  let q = [ expandCoreElement(s(i),m) | i in set inds s ]
  in return combine(q);
```

The combinations algorith, instanciated.

```
public combine :
  seq of TraceCoreDefinitionExpanded ==>
  seq of TraceDefinitionItemElement
combine(s) ==
  let l = [ s(i) | i in set inds s &
            is_TraceMethodApplyExpanded(s(i))]
  in return combineHelper(l);

public combineHelper :
```

```
  seq of TraceMethodApplyExpanded ==>
  seq of TraceDefinitionItemElement
combineHelper(s) ==
  if s = []
  then return []
  else let h = hd s,
           t = tl s,
           l = h.methodList
       in return concAllHelper(l,t);

public concAllHelper :
  seq of OmlTraceMethodApply *
  seq of TraceMethodApplyExpanded ==>
  seq of TraceDefinitionItemElement
concAllHelper(h,t) ==
  return conc [ concFirstHelper([[h(i)]],t) | i in set inds h];

public concFirstHelper :
  seq of (seq of OmlTraceMethodApply) *
  seq of TraceMethodApplyExpanded ==>
  seq of TraceDefinitionItemElement
concFirstHelper(t,s) ==
  if s = []
  then let meth = [ mk_TraceMethodApplyExpanded(t(i))
                  | i in set inds t ],
           elem = [ mk_TraceDefinitionItemElement(meth(i))
                  | i in set inds meth]
       in return elem
  else let head = hd s,
           headl = head.methodList,
           tail = tl s,
           mid = concatHelper(t,headl)
       in return concFirstHelper(mid,tail);

public concatHelper :
  seq of (seq of OmlTraceMethodApply) *
  seq of OmlTraceMethodApply ==>
  seq of (seq of OmlTraceMethodApply)
concatHelper(relPaths,t) ==
  return conc([mapValueHelper(relPaths(i),t)
              | i in set inds relPaths]);

public mapValueHelper :
  seq of OmlTraceMethodApply *
  seq of OmlTraceMethodApply ==>
  seq of (seq of OmlTraceMethodApply)
mapValueHelper(relPath,t) ==
  return [relPath ^ [t(i)] | i in set inds t];
```

The operations below are responsible for expanding a core element.

```
public expandCoreElement :
  OmlTraceDefinitionItem *
  TraceBindInformation ==>
  TraceCoreDefinitionExpanded
expandCoreElement(e,m) ==
  let t = e.getTest()
  in if (isofclass(OmlTraceBracketedDefinition,t))
     then return expandTraceBracketedDefinition(m,t)
     else return expandMethodApplyDefinition(m,t);


public expandTraceBracketedDefinition :
  TraceBindInformation *
  OmlTraceBracketedDefinition ==>
  TraceBracketedDefinitionExpanded
expandTraceBracketedDefinition(m,t) ==
  let d = t.getDefinition(),
      l = expandCore(d,m)
  in return mk_TraceBracketedDefinitionExpanded(l);

public expandCore :
  IOmlTraceDefinition *
  TraceBindInformation ==>
  TraceDefinitionExpanded
expandCore(d,m) ==
  cases true:
    (isofclass(OmlTraceDefinitionItem,d))
      -> return expandItemCore(d,m),
    (isofclass(OmlTraceSequenceDefinition,d))
      -> return expandSequence(d,m),
    others
      -> return expandChoice(d,m)
  end;

public expandItemCore :
  OmlTraceDefinitionItem *
  TraceBindInformation ==>
  TraceDefinitionItemExpanded
expandItemCore(a,n) ==
  let m = getInformation(a),
      q = traceBindInfoUnion(m,n),
      r = expandRegexpr(a),
      s = expandCoreDefinition(q,r)
  in return s;


public traceBindInfoUnion :
  TraceBindInformation *
  TraceBindInformation ==>
```

```
  TraceBindInformation
traceBindInfoUnion(s1,s2) ==

  let letIn = s1.letIn munion s2.letIn,
      letBe = s1.letBe munion s2.letBe,
      const = s1.constraints ^ s2.constraints
  in return mk_TraceBindInformation(letIn, letBe, const);

public expandMethodApplyDefinition :
  TraceBindInformation *
  OmlTraceMethodApply ==>
  TraceMethodApplyExpanded
expandMethodApplyDefinition(m,t) ==
  let v = t.getVariableName(),
      n = t.getMethodName(),
      a = t.getArgs(),
      s = combinations(m,a),
      r = toSeq(v,n,s)
  in return r;
```

The combinations algorithm, again instanciated.

```
public combinations :
  TraceBindInformation *
  seq of IOmlExpression ==>
  seq of (seq of IOmlExpression)
combinations(m,s) ==
  if hasNotOmlName(s)
  then return [s]
  else let l = [let name = evaluateExpression(s(e))
                in if name in set dom m.letBe
                   then seqModifyLetBe(s,e,getMapRange(name,m.letBe),
                                       m.constraints)
                   else [s ++ {e|-> getMapRange(name,m.letIn)}]
                | e in set inds s & isofclass(OmlName,s(e))]
       in return conc l;


public getMapRange :
  seq of char *
  map seq of char to IOmlExpression ==> IOmlExpression
getMapRange(key,m) == return m(key);


public hasNotOmlName :
  seq of IOmlExpression ==>
  bool
hasNotOmlName(s) ==
  if len s = 0
```

```
  then return true
  else return not isofclass(OmlName,hd(s)) and hasNotOmlName(tl(s));


public seqModifyLetBe :
  seq of IOmlExpression *
  nat1 *
  IOmlExpression *
  [seq of OmlExpression] ==>
  seq of (seq of IOmlExpression)
seqModifyLetBe(s,e,expr,c) ==
  cases true:
    (isofclass(OmlSetRangeExpression,expr))
      -> return getSetRange(s,e,expr),
    (isofclass(OmlSetEnumeration,expr))
      -> return getSetEnum(s,e,expr,c),
    (isofclass(OmlSequenceEnumeration,expr))
      -> return getSeqEnumeration(s,e,expr,c),
    others
      -> return []
  end;
```

Operations responsible for evaluating Oml expressions.

```
public getSeqEnumeration :
  seq of IOmlExpression *
  nat1 *
  OmlSequenceEnumeration *
  [seq of OmlExpression ] ==>
  seq of (seq of IOmlExpression)
getSeqEnumeration(s, e, expr,-) ==
  let x = expr.getExpressionList()
  in return [s ++ {e |-> x(i)} | i in set inds x];


public getSetRange :
  seq of IOmlExpression *
  nat1 *
  OmlSetRangeExpression ==>
  seq of (seq of IOmlExpression)
getSetRange(s, e, expr) ==
  let l = evaluateSetRange(expr),
      k = [ let n = new OmlNumericLiteral(l(i))
            in new OmlSymbolicLiteralExpression(n) | i in set inds l ]
  in return [ s ++ {e |-> k(i)} | i in set inds k];


public getSetEnum :
  seq of IOmlExpression *
```

```
  nat1 *
  OmlSetEnumeration *
  [seq of OmlExpression ] ==>
  seq of (seq of IOmlExpression)
getSetEnum(s, e, expr,-) ==
  let x = expr.getExpressionList()
  in return [s ++ {e |-> x(i)} | i in set inds x];


public toSeq :
  seq of char *
  seq of char *
  seq of (seq of IOmlExpression) ==>
  TraceMethodApplyExpanded
toSeq(v,n,s) ==
  let l = [new OmlTraceMethodApply(v,n,s(j)) | j in set inds s]
  in return mk_TraceMethodApplyExpanded(l);


public evaluateSeqEnumeration :
  OmlSequenceEnumeration ==>
  BasicType
evaluateSeqEnumeration(expr) ==
  let s = expr.getExpressionList()
  in return [ evaluateExpression(s(i)) | i in set inds s ];


public evaluateSetEnumeration :
  OmlSetEnumeration ==>
  BasicType
evaluateSetEnumeration(expr) ==
  let s = expr.getExpressionList()
  in return [ evaluateExpression(s(i)) | i in set inds s];


public evaluateMapEnumeration :
  OmlMapEnumeration ==>
  BasicType
evaluateMapEnumeration(expr) ==
  let s = expr.getMapletList()
  in return [ mk_(evaluateExpression(s(i).getDomExpression()),
                  evaluateExpression(s(i).getRngExpression()))
            | i in set inds s ];


public evaluateSetRange :
  OmlSetRangeExpression ==>
  BasicType
evaluateSetRange(expr) ==
  let l = evaluateExpression(expr.getLower()),
      u = evaluateExpression(expr.getUpper()),
```

```
      s = {l,...,u}
  in return [i | i in set s];

public evaluateBinary :
  OmlBinaryExpression ==>
  BasicType
evaluateBinary(expr) ==
  let l_expr = expr.getLhsExpression(),
      operat = expr.getOperator(),
      r_expr = expr.getRhsExpression()
  in return [evaluateExpression(l_expr),
             getValue(operat),
             evaluateExpression(r_expr)];

public evaluateName :
  OmlName ==>
  BasicType
evaluateName(expr) ==
  return expr.getIdentifier();

public evaluateExpression :
  IOmlExpression ==>
  BasicType
evaluateExpression(expr) ==
  cases true:
    (isofclass(OmlSymbolicLiteralExpression,expr))
     -> return getValueOfSymLit(expr),
    (isofclass(OmlSequenceEnumeration,expr))
     -> return evaluateSeqEnumeration(expr),
    (isofclass(OmlSetEnumeration,expr))
     -> return evaluateSetEnumeration(expr),
    (isofclass(OmlMapEnumeration,expr))
     -> return evaluateMapEnumeration(expr),
    (isofclass(OmlSetRangeExpression,expr))
     -> return evaluateSetRange(expr),
    (isofclass(OmlName,expr))
     -> return evaluateName(expr),
    (isofclass(OmlBinaryExpression,expr))
     -> return evaluateBinary(expr)
  end;

public getValueOfSymLit :
  OmlSymbolicLiteralExpression ==>
  BasicType
getValueOfSymLit(expr) ==
  let val = expr.getLiteral()
  in return getValue(val);

public getValue : IOmlLiteral ==> nat | real | bool
                                  | char | seq of char
getValue(lit) ==
```

```
  cases true:
    (isofclass(OmlNumericLiteral,lit))
     -> return getValueNumeric(lit),
    (isofclass(OmlRealLiteral,lit))
     -> return getValueReal(lit),
    (isofclass(OmlBooleanLiteral,lit))
     -> return getValueBoolean(lit),
    (isofclass(OmlCharacterLiteral,lit))
     -> return getValueChar(lit),
    (isofclass(OmlTextLiteral,lit))
     -> return getValueText(lit),
    (isofclass(OmlQuoteLiteral,lit))
     -> return getValueQuote(lit)
  end;

public getValueNumeric :
  OmlNumericLiteral ==>
  nat
getValueNumeric(lit) ==
  return lit.getVal();

public getValueReal :
  OmlRealLiteral ==>
  real
getValueReal(lit) ==
  return lit.getVal();

public getValueBoolean :
  OmlBooleanLiteral ==>
  bool
getValueBoolean(lit) ==
  return lit.getVal();

public getValueChar :
  OmlCharacterLiteral ==>
  char
getValueChar(lit) ==
  return lit.getVal();

public getValueText :
  OmlTextLiteral ==>
  seq of char
getValueText(lit) ==
  return lit.getVal();

public getValueQuote :
  OmlQuoteLiteral ==>
  seq of char
getValueQuote(lit) ==
  return lit.getVal();
```

```
public getValue :
  OmlBinaryOperator ==>
  nat
getValue(expr) ==
  return expr.getValue();
```

Whenever a test case has a regular expression, it is expanded using the following operations.

```
------------------------------------------------------------------
-- Expand Regular Expression  ------------------------------------
------------------------------------------------------------------

--Responsible for expanding the regexprs like +, *, ?, {n,m}
public expandRegexpr :
  OmlTraceDefinitionItem ==>
  seq of (seq of OmlTraceDefinitionItem)
expandRegexpr(i) ==
  return expandRegexprChoose(i);


public expandRegexprChoose :
  OmlTraceDefinitionItem ==>
  seq of (seq of OmlTraceDefinitionItem)
expandRegexprChoose(i) ==
  let r = i.getRegexpr()
  in if i.hasRegexpr()
     then return expandSymbol(i,r)
     else return [[i]];


public expandN2M :
  nat *
  nat *
  OmlTraceDefinitionItem *
  seq of OmlTraceDefinitionItem ==>
  seq of (seq of OmlTraceDefinitionItem)
expandN2M(n,m,s,o) ==
  if n <> m
  then return  [o] ^ expandN2M(n,m-1,s, o ^ [s])
  else if n = 0
       then return [[new OmlTraceDefinitionItem([],
                     new OmlTraceMethodApply([],[],[]),nil)]]
       else return [o];


public expandSymbol :
  OmlTraceDefinitionItem  *
```

```
  IOmlTraceRepeatPattern ==>
  seq of (seq of OmlTraceDefinitionItem)
expandSymbol(s,r) ==
  cases true:
    (isofclass(OmlTraceZeroOrMore,r))
      -> return expandSymbolZeroOrMore(s,r),
    (isofclass(OmlTraceOneOrMore,r))
      -> return expandSymbolOneOrMore(s,r),
    (isofclass(OmlTraceZeroOrOne,r))
      -> return expandSymbolZeroOrOne(s,r),
    (isofclass(OmlTraceRange,r))
      -> return expandSymbolRange(s,r)
  end;


public expandSymbolZeroOrMore :
  OmlTraceDefinitionItem *
  OmlTraceZeroOrMore ==>
  seq of (seq of OmlTraceDefinitionItem)
expandSymbolZeroOrMore(s,-) ==
  return expandN2M(0,zeroOrMoreMax,s,[s]);


public expandSymbolOneOrMore :
  OmlTraceDefinitionItem *
  OmlTraceOneOrMore ==>
  seq of (seq of OmlTraceDefinitionItem)
expandSymbolOneOrMore(s,-) ==
  return expandN2M(1,zeroOrMoreMax,s,[s]);


public expandSymbolZeroOrOne :
  OmlTraceDefinitionItem *
  OmlTraceZeroOrOne ==>
  seq of (seq of OmlTraceDefinitionItem)
expandSymbolZeroOrOne(s,-) ==
  return expandN2M(0,1,s,[s]);


public expandSymbolRange :
  OmlTraceDefinitionItem *
  OmlTraceRange ==>
  seq of (seq of OmlTraceDefinitionItem)
expandSymbolRange(s,t) ==
  let min = t.getLower().getVal(),
      max = getVal(min,t.getUpper())
  in return expandN2M(min,max,s,[s]);


public getVal :
  nat *
```

```
  [OmlNumericLiteral] ==>
  nat
getVal(min,n) ==
  if n = nil
  then return min
  else return n.getVal();
```

The operations below are responsible for gathering the information present in let expressions and save it for further usage.

```
------------------------------------------------------------------
-- Get Information - fill in trace bind information ---------------
------------------------------------------------------------------

--Responsible for collecting the let/let-be information for a
--structure that will be consulted when the core elements are to
--be expanded.
public getInformation :
  OmlTraceDefinitionItem ==>
  TraceBindInformation
getInformation(i) ==
  let b = i.getBind(),
      letBe = merge { getLetBeInfo(b(e)) |
         e in set inds b & isofclass(OmlTraceLetBeBinding,b(e)) },
      letIn = merge { getLetInfo(b(e)) |
         e in set inds b & isofclass(OmlTraceLetBinding,b(e)) },
      constraints = removeNil([ getConstraints(b(e)) |
         e in set inds b & isofclass(OmlTraceLetBeBinding,b(e)) ])
  in return mk_TraceBindInformation(letIn, letBe, constraints);


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


--------------------------------
-- Let Be Information ----------
--------------------------------

public getLetBeInfo :
  OmlTraceLetBeBinding ==>
  map seq of char to IOmlExpression
getLetBeInfo(b) ==
  return { extractBindingVariable(b)
           |->
           extractBindingExpression(b)
```

```
        }
pre isOfTypeSB(b);


public isOfTypeSB :
  OmlTraceLetBeBinding ==>
  bool
isOfTypeSB(b) ==
  let bind = b.getBind()
  in if isofclass(OmlSetBind,bind)
     then return isOfTypePattern(bind)
     else return false;


public isOfTypePattern :
  OmlSetBind ==>
  bool
isOfTypePattern(s) ==
  let p = s.getPattern(),
      v = p(1)
  in return isofclass(OmlPatternIdentifier,v)
pre len(s.getPattern()) = 1;


public extractBindingVariable :
  OmlTraceLetBeBinding ==>
  seq of char
extractBindingVariable(b) ==
  let bind = b.getBind()
  in return getVariable(bind)
pre is_(b.getBind(),OmlSetBind);


public getVariable :
  OmlSetBind ==>
  seq of char
getVariable(b) ==
  let p = b.getPattern(),
      v = p(1)
  in return getVariableName(v)
pre len(b.getPattern()) = 1
    and isofclass(OmlPatternIdentifier,b.getPattern()(1));

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

public extractBindingExpression :
  OmlTraceLetBeBinding ==>
```

```
  IOmlExpression
extractBindingExpression(b) ==
  let bind = b.getBind()
  in return getExpression(bind)
pre isofclass(OmlSetBind,b.getBind());

public getExpression :
  OmlSetBind ==>
  IOmlExpression
getExpression(b) ==
  return b.getExpression();

public getConstraints :
  OmlTraceLetBeBinding ==>
  [OmlExpression]
getConstraints(b) ==
  return b.getBest();

--------------------------------
-- Let Information ------------
--------------------------------

public getLetInfo :
  OmlTraceLetBinding ==>
  map seq of char to IOmlExpression
getLetInfo(b) ==
  let def_list = b.getDefinitionList()
  in return
      { getPatternId(def_list(e).getPattern())
        |->
        def_list(e).getExpression()
      | e in set inds def_list
        & isofclass(OmlPatternIdentifier,def_list(e).getPattern())
      };

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




------------------------------------------------
--------------- Input combinations ---------------
------------------------------------------------

public combinationsSeq :
  NamedTraceExpanded ==>
  seq of TraceDefinitionItemExpanded
```

```
combinationsSeq(d) ==
  let defs = d.defs
  in return combinationsHelper(defs);

public combinationsHelper :
  TraceDefinitionExpanded ==>
  seq of TraceDefinitionItemExpanded
combinationsHelper(t) ==
  cases true:
    (is_TraceDefinitionItemExpanded(t))
      -> return [ t ],
    (is_TraceSequenceDefinitionExpanded(t))
      -> return combineSequence(t),
    others
      -> return combineChoice(t)
  end;


public combineChoice :
  TraceChoiceDefinitionExpanded ==>
  seq of TraceDefinitionItemExpanded
combineChoice(s) ==
  let d = s.defs,
      l = [d(e) | e in set inds d]
  in return conc [cases true:
                  (is_TraceDefinitionItemExpanded(l(i)))
                  ->
                  aux(l(i)),
                  (is_TraceSequenceDefinitionExpanded(l(i)))
                  ->
                  combineSequence(l(i)),
                  others
                  ->
                  combineChoice(l(i))
                  end
                 | i in set inds l];

public combineSequence :
  TraceSequenceDefinitionExpanded ==>
  seq of TraceDefinitionItemExpanded
combineSequence(s) ==
  let d = s.defs
  in if d = []
     then return []
     else let news = [cases true:
                      (is_TraceDefinitionItemExpanded(d(i)))
                      ->
                      aux(d(i)),
                      (is_TraceSequenceDefinitionExpanded(d(i)))
                      ->
                      combineSequence(d(i)),
```

```
                      (is_TraceChoiceDefinitionExpanded(d(i)))
                      ->
                      combineChoice(d(i))
                      end
                     | i in set inds d]
          in return combineSequenceHelper(news);


public aux :
  TraceDefinitionItemExpanded ==>
  seq of (TraceDefinitionItemExpanded)
aux(t) ==
  let l = t.item_list
  in return [ mk_TraceDefinitionItemExpanded([l(i)])
            | i in set inds l];


public combineSequenceHelper :
  seq of (seq of TraceDefinitionItemExpanded) ==>
  seq of TraceDefinitionItemExpanded
combineSequenceHelper(s) ==
  if s = [] then return [] else
  let h = hd s,
      t = tl s,
      l = [ h(i).item_list | i in set inds h ]
  in return concAll(l, t);

public concAll :
  seq of (seq of TraceDefinitionItemElement) *
  seq of (seq of TraceDefinitionItemExpanded) ==>
  seq of TraceDefinitionItemExpanded
concAll(h,t) ==
  let l = conc [concFirst([h(i)],t) | i in set inds h]
  in return l;

public concFirst :
  seq of (seq of TraceDefinitionItemElement) *
  seq of (seq of TraceDefinitionItemExpanded) ==>
  seq of TraceDefinitionItemExpanded
concFirst(t,s) ==
  if s = []
  then return [ mk_TraceDefinitionItemExpanded(t(i))
              | i in set inds t] else
  let head = hd s,
      headl = [head(i).item_list | i in set inds head],
      tail = tl s,
      mid = concat(t,headl)
  in return concFirst(mid,tail);


public concat :
```

```
  seq of (seq of TraceDefinitionItemElement) *
  seq of (seq of TraceDefinitionItemElement) ==>
  seq of (seq of TraceDefinitionItemElement)
concat(relPaths,t) ==
  return conc [ mapValue(relPaths(i),t) | i in set inds relPaths ];


public mapValue :
  seq of TraceDefinitionItemElement *
  seq of (seq of TraceDefinitionItemElement) ==>
  seq of (seq of TraceDefinitionItemElement)
mapValue(relPath,t) ==
  return [ relPath ^ t(i) | i in set inds t];

end Expanded
```

D.2 pp.vpp

```
class PP

types

public String = seq of char;

operations

public ppBasicType :
  Expanded'BasicType ==>
  seq of (seq of char)
ppBasicType(b) ==
  if is_(b,bool)
  then return [bool2string(b)]
  else if is_(b,char) or is_(b,real) or is_(b,nat)
       then return [[b]]
       else if is_(b,seq of Expanded'BasicType)
            then return [conc [conc ppBasicType(b(i))
                               | i in set inds b]]
            else return [conc ppBasicType(b(i).#1) ^
                         conc ppBasicType(b(i).#2)
                        | i in set inds b];

public bool2string :
  bool ==>
  seq of char
bool2string(b) ==
  if b=true then return "true"
            else return "false";
```

```
public ppMethApplyExp :
  Expanded‘TraceMethodApplyExpanded ==>
  seq of Expanded‘BasicType
ppMethApplyExp(s) ==
  let m = s.methodList
  in return [ let q = m(i).getArgs()
              in [ evaluateExpression(q(j))
                 | j in set inds q]
            | i in set inds m];

public ppCoreDefExp :
  Expanded‘TraceBracketedDefinitionExpanded ==>
  seq of (seq of Expanded‘BasicType)
ppCoreDefExp(c) ==
  return ppDefExp(c.definition);

public ppDefExp :
  Expanded‘TraceDefinitionExpanded ==>
  seq of (seq of Expanded‘BasicType)
ppDefExp(d) ==
  cases true:
  (is_Expanded‘TraceDefinitionItemExpanded(d))
  ->
  ppDefExpanded(d),
  (is_Expanded‘TraceSequenceDefinitionExpanded(d))
  ->
  ppSeqDefExp(d),
  others
  ->
  ppChoiceDefExp(d)
  end;

public ppSeqDefExp :
  Expanded‘TraceSequenceDefinitionExpanded ==>
  seq of (seq of Expanded‘BasicType)
ppSeqDefExp(s) ==
  let d = s.defs
  in return conc [ppDefExp(d(i)) | i in set inds d];

public ppChoiceDefExp :
  Expanded‘TraceChoiceDefinitionExpanded ==>
  seq of (seq of Expanded‘BasicType)
ppChoiceDefExp(s) ==
  let d = s.defs
  in return conc [ppDefExp(d(i)) | i in set inds d];

ppTraceDefsExpanded :
  Expanded‘TraceDefinitionsExpanded ==>
  seq of Expanded‘BasicType
ppTraceDefsExpanded(d) ==
  let x = d.defs
```

```
  in return conc [ [x(i).name] ^ ppDefExp(x(i).defs)
                 | i in set inds x];

public ppSeqDefItemExpanded :
  seq of Expanded‘TraceDefinitionItemExpanded ==>
  seq of (seq of Expanded‘BasicType)
ppSeqDefItemExpanded(d) ==
  return [ ppDefExpanded(d(i)) | i in set inds d];


public ppDefExpanded :
  Expanded‘TraceDefinitionItemExpanded ==>
  seq of (seq of Expanded‘BasicType)
ppDefExpanded(d) ==
  let l = d.item_list
  in return conc [ppTraceDefItemElement(l(i)) | i in set inds l];


public ppTraceDefItemElement :
  Expanded‘TraceDefinitionItemElement ==>
  seq of (seq of Expanded‘BasicType)
ppTraceDefItemElement(d) ==
  let t = d.test
  in return ppTraceCoreDefExpanded(t);


public ppTraceCoreDefExpanded :
  Expanded‘TraceCoreDefinitionExpanded ==>
  seq of (seq of Expanded‘BasicType)
ppTraceCoreDefExpanded(c) ==
  return ppTraceMethApplyExpanded(c);


public ppTraceMethApplyExpanded :
  Expanded‘TraceMethodApplyExpanded ==>
  seq of Expanded‘BasicType
ppTraceMethApplyExpanded(s) ==
  let m = s.methodList
  in return
  [ let q = m(i).getArgs(),
        v = m(i).getVariableName(),
        meth = m(i).getMethodName()
    in if v = [] then [] else
       v ^ "." ^ meth ^ "(" ^
           [ evaluateExpression(q(j))
           | j in set inds q] ^ ")"
   | i in set inds m];


--------------------------------------
-- Get values of variables from the map
```

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

public evaluateExpression :
  OmlSymbolicLiteralExpression ==>
  Expanded‘BasicType
evaluateExpression(expr) ==
  return getValueOfSymLit(expr);

public evaluateExpression :
  OmlSequenceEnumeration ==>
  seq of Expanded‘BasicType
evaluateExpression(expr) ==
  let s = expr.getExpressionList()
  in return [ evaluateExpression(s(i)) | i in set inds s ];

public evaluateExpression :
  OmlSetEnumeration ==>
  seq of Expanded‘BasicType
evaluateExpression(expr) ==
  let s = expr.getExpressionList()
  in return [ evaluateExpression(s(i)) | i in set inds s];

public evaluateExpression :
  OmlMapEnumeration ==>
  seq of (Expanded‘BasicType * Expanded‘BasicType)
evaluateExpression(expr) ==
  let s = expr.getMapletList()
  in return [ mk_(evaluateExpression(s(i).getDomExpression()),
              evaluateExpression(s(i).getRngExpression()))
            | i in set inds s ];

public evaluateExpression :
  OmlSetRangeExpression ==>
  seq of nat
evaluateExpression(expr) ==
  let l = evaluateExpression(expr.getLower()),
      u = evaluateExpression(expr.getUpper()),
      s = {l,...,u}
  in return [i | i in set s];

public evaluateExpression :
  OmlName ==>
  seq of char
evaluateExpression(name) ==
  return name.getIdentifier();

public evaluateExpression :
  OmlBinaryExpression ==>
  seq of Expanded‘BasicType
evaluateExpression(expr) ==
  let l_expr = expr.getLhsExpression(),
```

```
      operat = expr.getOperator(),
      r_expr = expr.getRhsExpression()
  in return [evaluateExpression(l_expr), getValue(operat),
             evaluateExpression(r_expr)];

public getValueOfSymLit :
  OmlSymbolicLiteralExpression ==>
  Expanded‘BasicType
getValueOfSymLit(expr) ==
  let val = expr.getLiteral()
  in return getValue(val);

public getValue :
  OmlNumericLiteral ==>
  nat
getValue(lit) ==
  return lit.getVal();

public getValue :
  OmlRealLiteral ==>
  real
getValue(lit) ==
  return lit.getVal();

public getValue :
  OmlBooleanLiteral ==>
  bool
getValue(lit) ==
  return lit.getVal();

public getValue :
  OmlCharacterLiteral ==>
  char
getValue(lit) ==
  return lit.getVal();

public getValue :
  OmlTextLiteral ==>
  seq of char
getValue(lit) ==
  return lit.getVal();

public getValue :
  OmlQuoteLiteral ==>
  seq of char
getValue(lit) ==
  return lit.getVal();

public getValue :
  OmlBinaryOperator ==>
  nat
```

```
getValue(expr) ==
  return expr.getValue();

end PP
```

D.3 Filtering.vpp

```
class Filtering

instance variables

public testCases : TestCases;
public totTestCases : nat;
public passTestCases : nat;
public seqOfSelected : seq of (seq of char);
public seqOfDeleted : seq of (seq of char);
public tb : ToolBox;

types

public Output = (nat * seq of char);

public TestCases ::
  allTestCases : map (seq of Expanded‘BasicType) to Output
  failedTestCases : map (seq of Expanded‘BasicType) to nat;
  -- seq of char is a test case
  -- The nat can be
  -- 0 : initial value
  -- 1 : the test case had a pass verdict
  -- 2 : the test case had a fail verdict

public Statistics ::
  initText : seq of char
  selected : seq of (seq of char)
  deletedText : seq of char
  deleted : seq of (seq of char)
totSelectedText : seq of char
totSelected : nat
totFailedText : seq of char
totFailed : nat
totDeletedText : seq of char
totDeleted : nat
percentText : seq of char
percentFailed : real
percentDeletedText : seq of char
percentDeleted : real
;

operations
```

```
public Filtering :
  seq of (seq of Expanded‘BasicType) *
  seq of (seq of char) ==>
  Filtering
Filtering(t, specs) == (
testCases.allTestCases := {t(i)|->mk_(0,"") | i in set inds t};
testCases.failedTestCases := {|->};
totTestCases := 0;
passTestCases := 0;
seqOfSelected := [];
seqOfDeleted := [];
totTestCases := card (dom testCases.allTestCases);
 tb := new ToolBox(specs);
);

--------------------------------------------------------------------
---------------------- Pretty Printing ---------------------------
--------------------------------------------------------------------


public ppTestCases :
  () ==>
  Statistics
ppTestCases() ==
  let f = filter(),
      b = insertValuesSelected(),
      c = insertValuesDeleted(),
      firstFailed = len(seqOfSelected) - passTestCases
  in return mk_Statistics(
    "Executed test cases: \n",
    [seqOfSelected(i) ^ "\n" | i in set inds seqOfSelected],
    "\n Failed test cases:\n",
    [seqOfDeleted(i) ^ "\n" | i in set inds seqOfDeleted],
    "\n\n Statistics:\n Number of selected test cases: ",
    len(seqOfSelected),
    "\n Number of failed test cases: ",
    len(seqOfDeleted),
    "\n Number of deleted test cases: ",
    len(seqOfDeleted) - firstFailed,
    "\n Percentage of failed test cases: ",
    ((len(seqOfDeleted))/totTestCases)*100,
    "Percentage of deleted test cases: ",
    ((len(seqOfDeleted) - firstFailed)/totTestCases)*100
    );

public insertValuesSelected :
  () ==>
  bool
insertValuesSelected() ==
  return forall i in set dom testCases.allTestCases
```

```
                & insertSelected(i);

public insertSelected :
  seq of char ==>
  bool
insertSelected(i) ==
  (
  seqOfSelected := seqOfSelected ^ [i];
  return true;
  );

public insertValuesDeleted :
  () ==>
  bool
insertValuesDeleted() ==
  return forall i in set dom testCases.failedTestCases
                & insertDeleted(i);

public insertDeleted :
  seq of char ==>
  bool
insertDeleted(i) ==
  (
  seqOfDeleted := seqOfDeleted ^ [i];
  return true;
  );

public filter :
  () ==>
  bool
filter() ==
  let s = dom testCases.allTestCases
  in return forall i in set s & test(i);

public test :
  seq of Expanded‘BasicType ==>
  bool
test(t) ==
  let res = tb.vdmToolsCall(t)
  in if res.verdict = 2
     then return maybeRemove(t)
     else (testCases.allTestCases(t) := mk_(1,res.output);
           passTestCases := passTestCases+1;
           return true;
          );

public maybeRemove :
  seq of Expanded‘BasicType ==>
  bool
maybeRemove(t) ==
  let m = {i|->2 | i in set dom testCases.allTestCases
```

```
                    & prefix(t,i)}
  in (testCases.allTestCases := dom(m) <-: testCases.allTestCases;
  testCases.allTestCases := testCases.allTestCases ++
                            {t|->mk_(2,"Failed")};
  testCases.failedTestCases := testCases.failedTestCases ++ m;
  return true;
  );

-- prefix : failed test ==> is this test case a prefix of t? ==> Y/N
public prefix :
  seq of Expanded'BasicType *
  seq of Expanded'BasicType ==>
  bool
prefix(t,i) ==
  let m = [t(j)=i(j) | j in set inds t & len t <= len i]
  in return conjoin(m);

public conjoin :
  seq of bool ==>
  bool
conjoin(s) ==
  if s = [] then return true
            else return hd(s) and conjoin(tl(s));

end Filtering
```

D.4 Toolbox.vpp

```
class ToolBox

types

public interpreterResult ::
        verdict : nat
        output : seq of char
inv x == x.verdict in set {0,1,2};

instance variables

public specs: seq of (seq of char);

operations

public ToolBox :
  seq of (seq of char) ==>
  ToolBox
ToolBox(sp) ==
  ( specs := sp;
  );
```

```
public vdmToolsCall :
  seq of char ==>
  interpreterResult
vdmToolsCall(-) ==
  return mk_interpreterResult(2,"Failed");

end ToolBox
```

D.5 CTesting.vpp

```
class CTesting

instance variables

public rex : Expanded;
public pp   : PP;
public erex : seq of Expanded‘TraceDefinitionItemExpanded;
public pp_rex : seq of (seq of Expanded‘BasicType);
public filter : Filtering;
public stat : Filtering‘Statistics;

operations

public CTesting :
  OmlTraceDefinitions *
  seq of (seq of char) ==>
  CTesting
CTesting(r, specs) ==
  (
  rex    := new Expanded();
  pp     := new PP();
  erex   := rex.expandTraces(r);
  pp_rex := pp.ppSeqDefItemExpanded(erex);
  filter := new Filtering(pp_rex, specs);
  stat   := filter.ppTestCases();
  );

end CTesting
```

List of Figures

List of Tables