

VDMTools

VDMTools User Manual (VDM++)
ver.1.0



How to contact:

<http://fmvdm.org/>

VDM information web site(in Japanese)

<http://fmvdm.org/tools/vdmttools>

VDMTools web site(in Japanese)

inq@fmvdm.org

Mail

VDMTools User Manual (VDM++) 1.0

— Revised for VDMTools v9.0.6

© COPYRIGHT 2016 by Kyushu University

The software described in this document is furnished under a license agreement.
The software may be used or copied only under the terms of the license agreement.

This document is subject to change without notice

Contents

1	Introduction	1
2	VDMTools Overview	4
3	A Guided Tour of the VDMTools	7
3.1	Creating Input to VDMTools	7
3.2	Starting the VDM++ Graphical User Interface	8
3.3	On-Line Help	8
3.4	Menus, Toolbars and Subwindows	9
3.5	Configuring your Project	10
3.6	Syntax Checking your VDM Specification	12
3.6.1	Parsing the specification	12
3.6.2	Correcting syntax errors	13
3.7	Type Checking your VDM Specification	15
3.8	Validating your Specification	19
3.8.1	Evaluating expressions using the interpreter	19
3.8.2	Setting breakpoints	22
3.8.3	Dynamic type checking	24
3.8.4	Checking Integrity Properties	27
3.8.5	Multi-threaded models	31
3.9	Introducing Systematic Testing	31
3.10	Pretty Printing	33
3.11	Generating Code	34
3.12	The VDMTools API	35
3.13	Exiting VDMTools	35
4	The VDMTools Reference Manual	36
4.1	The Overall Graphical User Interface	36
4.1.1	Project handling	37
4.1.2	Operations on specifications	40
4.1.3	The log window, error list and source window	41
4.1.4	Editing files	43
4.1.5	Using the interpreter	44
4.1.6	On-line help	44
4.2	The Overall Command Line Interface	44
4.2.1	Initialisation file	46
4.3	The Syntax Checker	48
4.3.1	The graphical user interface	48
4.3.2	Format of syntax errors	49
4.3.3	The command line interface	49



4.3.4	The Emacs interface	50
4.4	The Type Checker	52
4.4.1	The graphical user interface	53
4.4.2	Format of type errors and warnings	53
4.4.3	The command line interface	55
4.4.4	The Emacs interface	56
4.5	The Interpreter and Debugger	58
4.5.1	The graphical user interface	58
4.5.2	Standard libraries	66
4.5.3	The command line interface	67
4.5.4	The Emacs interface	68
4.5.5	Scheduling of threads	74
4.6	The Integrity Examiner	75
4.7	The Pretty Printer	77
4.7.1	The graphical user interface	77
4.7.2	The command line interface	78
4.7.3	The Emacs interface	80
4.8	The VDM++ to C++ Code Generator	81
4.8.1	The graphical user interface	81
4.8.2	The command line interface	81
4.8.3	The Emacs interface	82
4.9	The VDM++ to Java Code Generator	83
4.9.1	The graphical user interface	83
4.9.2	The command line interface	85
4.9.3	The Emacs interface	85
4.10	Systematic Testing of VDM models	86
4.10.1	Preparing the test coverage file	87
4.10.2	Updating the test coverage file	87
4.10.3	Producing the test coverage statistics	88
4.10.4	Test coverage example using \LaTeX	89
	Glossary	96
	A Information Resources on VDM Technology	98
	B Combining VDM++ and \LaTeX	100
B.1	Format of a Specification File	100
B.2	Setting up a \LaTeX Document	100

C	Setting up your VDMTools Environment	104
C.1	Interface Options	104
C.2	Multilingual Support	105
C.3	UML Link support	107
D	The Emacs Interface	108
E	Test Scripts for the Sorting Example	109
E.1	The Windows/DOS Platform	109
E.2	The UNIX Platforms	110
F	Troubleshooting Problems with Microsoft Word	112
G	Format for Priority File	113
	Index	114

1 Introduction

VDMTools is a set of tools that allows you to develop and analyse precise models of computing systems. When used in the early stages of system development, these models can serve as system specifications, or as an aid in checking the consistency and completeness of user requirements. The models are expressed either in the ISO VDM-SL standard language [13] or in the object-oriented formal specification language VDM++ [4, 12]. This manual describes the VDM++ Toolbox, which provides a range of tools for automatic checking and validation of models expressed in VDM++ prior to implementation. These range from traditional syntax and type checking tools to a powerful interpreter that executes models on request and performs automatic consistency checking during execution. The execution facilities support the use of testing techniques in early analysis and design and allow execution of entire test suites in line with established software engineering practice. Moreover, the interpreter enables interactive debugging of models by setting breakpoints, stepping through statements and expression evaluations, inspecting the call stack, and checking the values of variables in scope.

This document provides an introduction and reference manual to the VDM++ Toolbox (called the Toolbox in the remainder of the document). The VDM++ language is described in the separate language reference manual [4]. In the remainder of this manual we use the term *specification* to refer to any model constructed in the language for whatever purpose.

VDM Input Formats

The Toolbox supports VDM++ specifications embedded in either Microsoft Word or L^AT_EX documents so that it is possible to analyse specifications without having to extract them into a special file. We recommend the use of either one of these two approaches as an excellent way of combining the model of a system with its documentation. Having just one version of the specification helps to avoid inconsistencies arising between working and documented versions of the specification. If you would rather not use Word or L^AT_EX, you can of course write specifications as clear text in plain ASCII files using your preferred text editor.

The Toolbox supports input documents in a range of different languages and scripts. Appendix C.2 explains how to configure the Toolbox for different scripts.

If you use Microsoft Word to write your VDM++ specifications, you should save the documents containing specifications in *rich text format* (RTF). The Toolbox

distribution contains example files in this format. Throughout this manual, you will see examples using files from the Toolbox distribution. The names of such example files are followed by the extension “.rtf”, indicating that they are in rich text format.

In this manual we will normally introduce features of the Toolbox using Word and RTF. If you use the L^AT_EX text processing system to write your specifications, then note that the Toolbox expects input containing L^AT_EX commands mixed with VDM specifications using the style and format described in Appendix B. The Toolbox distribution also contains example files in this format, indicated by the filename extension “.vpp” rather than “.rtf”. Thus, if an example refers to a file called `sort.rtf`, you should instead use the file `sort.vpp`. References to a directory structure are shown throughout this manual in the form `examples/sort.vdm` (i.e. with a forward slash) unless the reference is only relevant under Windows in which case it is shown as `examples\sort.vdm` (i.e. with a backward slash).

If you prefer to write specifications as plain text ASCII files, note that the only way to incorporate explanatory text into your specification is by means of the VDM++ comment syntax, described in the Language Manual. Files prepared in this format are normally given a “.vpp” extension.

Using This Manual

This manual is divided into three parts. Sections 2 and 3 provide an overview of the various tools in the Toolbox and a “hands-on” tutorial introduction to using the Toolbox via its graphical user interface. Before working through this part of the manual the Toolbox should be installed and the environment variables required should be set (see Appendix C). The installation of the Toolbox is described in the document [3]. As you work through Section 3, you will get to know the various tools and control commands available to you.

The second part of the manual (Section 4) is a reference guide covering all the features of the Toolbox systematically. All three available interfaces – the command line interface, the Emacs interface and the graphical user interface – are described for each feature.

The third part of the manual consists of appendices on a range of topics. Appendix A includes pointers to information resources for VDM, including internet sites, project descriptions, technical papers and books. Appendix B explains how you merge text and specification in L^AT_EX documents.

Appendix [C](#) describes which environment variables and options can be set for the Toolbox. Appendix [D](#) describes the Emacs interface. Appendix [E](#) presents a few test scripts used for systematic testing of the sorting specification which is used as a running example in this manual. Appendix [F](#) offers some possible solutions to common problems encountered when using the Toolbox in conjunction with Microsoft Word. And Appendix [G](#) describes the format for defining priority files for use with the interpreter.

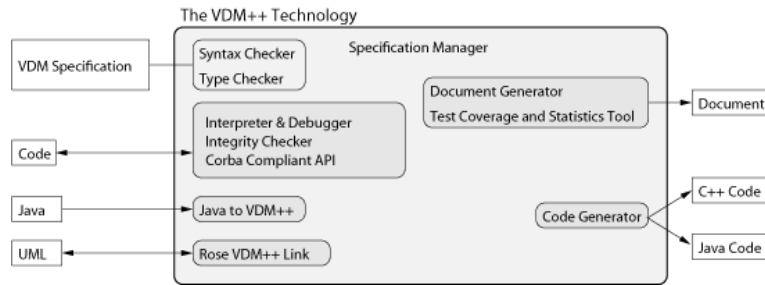


Figure 1: Overview of **VDMTools**

2 VDMTools Overview

A VDM++ specification is a document which aims to describe the properties of a system in a precise way. The specification can be distributed among several files in the input formats described in Section 1. Figure 1 provides an overview of the functionality of the Toolbox and its additional features. The various tools are described below:

Specification Manager The specification manager keeps track of the status of classes in the specification, which may be spread across several files.

Syntax Checker: The syntax checker checks whether the syntax of your VDM++ specification is correct with respect to the definition of the VDM++ language. If the syntax is accepted it gives access to the other tools in the Toolbox.

Type Checker The type checker contains a powerful type inference mechanism which identifies mis-uses of values and operators and which can also show all places where run-time errors could occur.

Interpreter and Debugger The interpreter allows you to execute all the executable constructs in VDM++. These range from simple value builders like set comprehension and sequence enumeration to more advanced constructs like exception handling, lambda expressions, loose expressions and pattern matching, or even multithreaded models. One of the benefits of executing specifications is that testing techniques can be used to assist in their validation. In the development process small or large parts of a specification can be executed to enhance the designer's knowledge of and confidence in the specification. Furthermore, an executable specification can form a running prototype.

A source-level debugger is an essential aid when working with executable specifications. The VDM++ debugger supports the functionality found in debuggers for ordinary programming languages, including setting break-points, stepping, inspection of variables defined in the scope, and inspection of the call stack.

Integrity Examiner The integrity examiner extends the static checking capabilities of the VDM++ Toolbox by scanning through specifications checking for potential sources of internal inconsistencies or integrity violations. The checks include the violation of data type invariants, preconditions, post-conditions, sequence bounds and map domains. Each *integrity property* is presented as a VDM++ expression which should evaluate to true – if it instead evaluates to false this indicates that there is a potential problem with the corresponding part of the specification.

Test Facility The test facility allows you to exercise your specification using a predefined set of tests called a *test suite*. Test coverage information can be automatically recorded during execution of a test suite and presented back to the specifier, indicating which parts of the specification are most frequently evaluated and which parts have not been covered at all. The test coverage information is displayed directly in the source file which can be a Microsoft Word or L^AT_EX document depending upon the input format used.

Automatic Code Generator The Toolbox supports automatic generation of C++ and Java code from VDM++ specifications¹, helping to achieve consistency between specification and implementation. The code generator produces fully executable code for 95% of all VDM++ constructs, and there are facilities for including user-defined code for non-executable parts of the specification. Once a specification has been tested, the code generator can be applied to obtain a rapid implementation automatically. The use of the C++ Code Generator is described in the document [7] and the use of the Java Code Generator is described in the document [8].

Corba Compliant API The Toolbox provides a Corba compliant Application Programmer Interface (API) which allows other programs to access a running Toolbox. This enables external control of the the Toolbox components such as the type checker, interpreter and debugger. The API allows any code such as a graphical front-end or existing legacy code to control the Toolbox.

Rose-VDM++ Link The Rose-VDM++ link integrates UML and VDM++. It provides a bi-directional translation which gives a tight coupling between

¹You must have a separate Code Generator license to use this facility.

the Toolbox and Rational Rose. Hence the link supports round trip engineering between UML and VDM++, where the graphical notation is used to provide the structural, diagrammatic overview of a model while the formal notation is used to provide the detailed functional behaviour. The use of the Rose-VDM++ link is described in the document [\[2\]](#).

Java to VDM++ Translator This feature allows existing legacy Java applications to be reverse engineered to VDM++. Analysis of the application can then be performed at the VDM++ level and new features can be specified. Finally, the new specification can be translated back to Java. The use of the Java to VDM++ Translator is described in [\[1\]](#).

3 A Guided Tour of the VDMTools

This section provides a “guided tour” of the Toolbox. If you are new to the principles of system modelling in VDM, we recommend that you should first read either “*Modelling Systems: Practical Tools and Techniques in Software Development*” [11], by John Fitzgerald and Peter Gorm Larsen or “*Validated Designs for Object-oriented Systems*” [12]. These are both tutorial books which includes many examples built around VDM specifications which can be explored using the Toolbox. [11] is using the ISO standard VDM-SL notation whereas [12] is using the object-oriented extension called VDM++. If you do have some knowledge about these general concepts, but are unfamiliar with the object-oriented extensions in VDM++, we recommend that you review the VDM++ language reference manual “*The VDM++ Specification Language*” [4].

3.1 Creating Input to VDMTools

In order to use the Toolbox it is necessary to produce a VDM++ specification. In this section we illustrate how to do that using Microsoft Word in the rich text format (RTF) on a simple sorting example. If you alternatively prefer using VDM++ combined with L^AT_EX you should consult Appendix B². In the remainder of this section we assume some basic familiarity with Microsoft Word.

Start Microsoft Word by selecting it from the programs entry in the Windows setup under Windows. Open the `vpphome/examples/sort/MergeSort.rtf` file from the Toolbox distribution. Reading through this file, you will see that the document is a mixture of explanatory text and a formal model in VDM++. All the formal parts are written in the style `VDM`. You may not change the formatting of the text in the VDM style directly in the source text. The pretty-printer will put VDM keywords in the boldface font anyway. If you wish, you can modify the appearance of this style, so long as the style’s name is not changed: the Toolbox will only analyse those parts of the document written in the `VDM` style.

A definition of the styles which are used by the Toolbox inside Microsoft Word can be found in the `VDM.dot` file from the Toolbox distribution. This file can be copied to your template directory (`C:\Program Files\Microsoft Office\Templates` normally) so that these style definitions will be included if you select this template when a new document is started (there are also various ways in which the definitions can be copied into the template file you normally use).

²It is also possible to use plain ASCII VDM++.

Now look at the end of the `MergeSort.rtf` document. You will see that the name of the class `MergeSort` is written in the style `VDM_TC_TABLE`. We will come back to the usage of this when we discuss how to record and display test coverage information. The styles `VDM_COV` and `VDM_NCOV` are also used in connection with test coverage information. We will also come back to these styles later.

If you wish to gain more experience with using Microsoft Word for producing your input to the Toolbox we recommend that you try to read in some of the other examples from the Toolbox distribution after completing this guided tour.

3.2 Starting the VDM++ Graphical User Interface

The Toolbox is normally used via its graphical user interface. Before starting this interface, VDM source files should be copied into a working directory. The Toolbox distribution contains a specification of different sorting algorithms, a presentation of which can be found in the technical report [6]. During this guided tour we will use this sorting specification as our running example, so copy the directory `vpphome/examples/sort` from the Toolbox distribution and `cd` to it. This will enable you to try the tools in the Toolbox directly on your computer while you are following the tour.

The Toolbox is started by selecting it from the “Program Files” entry in the Windows start menu or with the command `vppgde` on Unix platforms. The Toolbox will start up as shown in Figure 2. This window is called the *main window* of the Toolbox.

3.3 On-Line Help

On-line help for the Toolbox and the interface in general can be accessed through the **Help** toolbar or the **Help** menu. Currently only the following very limited help is available:

About (): Displays the version number of the Toolbox.

aboutqt (): Displays information about and a reference to Qt, the multiplatform C++ GUI toolkit which the Toolbox interface uses.

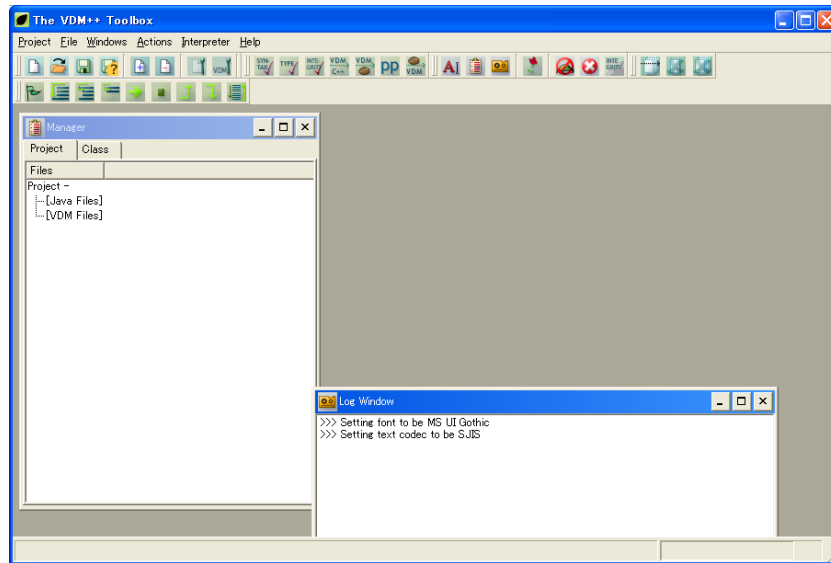


Figure 2: Graphical User Interface Startup

3.4 Menus, Toolbars and Subwindows

The top of the main window consists of a menu line with six pull-down menus:

Project: A project consists of a collection of file names that together form a VDM++ specification. Under this menu heading it is possible to open and save projects, to configure (add files to and remove files from) projects, and to create new projects. This is also the place from which to exit the Toolbox and to set options for the various tools in the Toolbox, for example to govern the level of type checking.

File: Here you can invoke a file editor for making corrections to your specification and also remove displays of source files generated by the Toolbox when it reports errors.

Windows: Controls to determine which windows are displayed in the bottom pane of the main window. Each menu item toggles opening/closing of a particular window.

Actions: This offers the various actions that can be applied to a specification: syntax and type checking, generation of integrity properties, code generation, translation from Java to VDM++, and pretty printing.

Interpreter: This offers functions for controlling the interpreter (see Section 3.8.1).

Below this menu line are six³ toolbars comprising buttons which offer the same actions⁴.

Finally, the lower pane of the main window is used to display various subwindows which either present information about the status of the current project or offer interfaces to tools within the Toolbox. The available windows are as follows:

Manager Displays the current status of the current project. It consists of two parts:

Project View This shows a tree representation of the contents of the project comprising the files in the project and (only after successfully syntax checking the file) the classes declared in each file.

Class View This offers both a **VDM View** and a **Java View**, which display the status of each of the individual VDM++ classes or Java files in the project respectively.

Source Window Displays the part of the source specification in which the error currently selected in the **Error List** was discovered.

Log Window Displays messages from the Toolbox.


Interpreter Window The interface to the interpreter.

Error List Reports errors found by the Toolbox.

Integrity Properties Window Displays the integrity properties that have been generated for the specification.

When the Toolbox is started, only the **Manager** and **Log Window** is open.

3.5 Configuring your Project

First you need to configure the Toolbox by indicating which files (in your desired input format) are to be analysed. For this purpose you can select the action **Add File to Project** on the **Project** menu or simply press the  (**Add Files**) button on

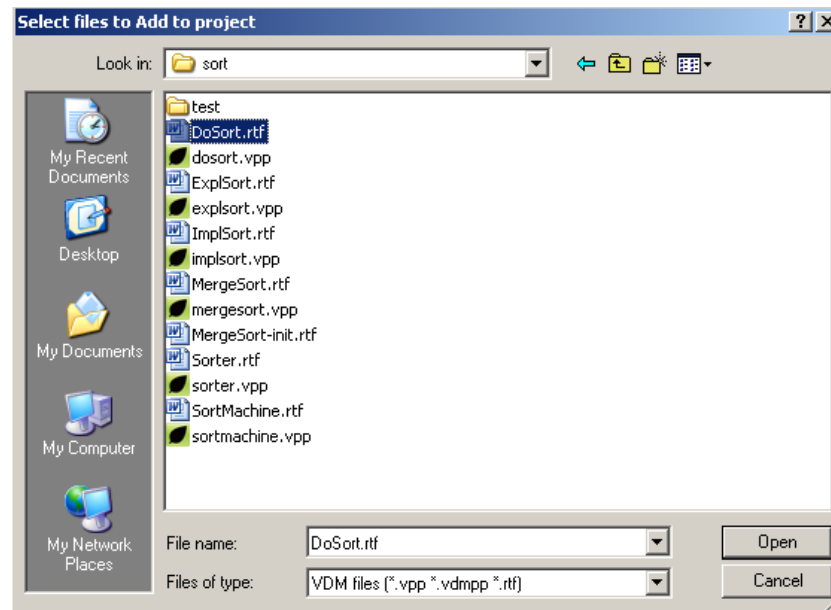


Figure 3: Adding Files to a Project

the (Project Operations) toolbar⁵. The dialog box shown in Figure 3 will then appear.

Mark the six `.rtf` files (minus the `MergeSort.rtf` file) by holding down the `Ctrl` key and clicking the left-hand mouse button on each of the files in turn, then press the “Open” button. These files will then be included in the project. You can also add a single file to a project by double clicking the left-hand mouse button on it (but note that this also closes the dialog box so it is not an efficient way of adding a number of files), and you can also mark a list of files at the same time by selecting the first and last files in the list (in either order), holding down the `Shift` key while making the second selection. Note that `MergeSort-init.rtf` contains a number of errors for illustration purposes in this guided tour.

The six `.rtf` files will now appear in the Project View of the Manager in the main Toolbox window as shown in Figure 4.

³When the Toolbox is started, only the three which correspond to the first, third and fourth menus are displayed open; the other three are displayed in iconised form above them.

⁴Except that the function for exiting from the toolbox is only available on the Project menu.

⁵In the remainder of this guided tour we concentrate on interactions via the toolbar buttons. You can of course always use the equivalent menu item if you prefer.

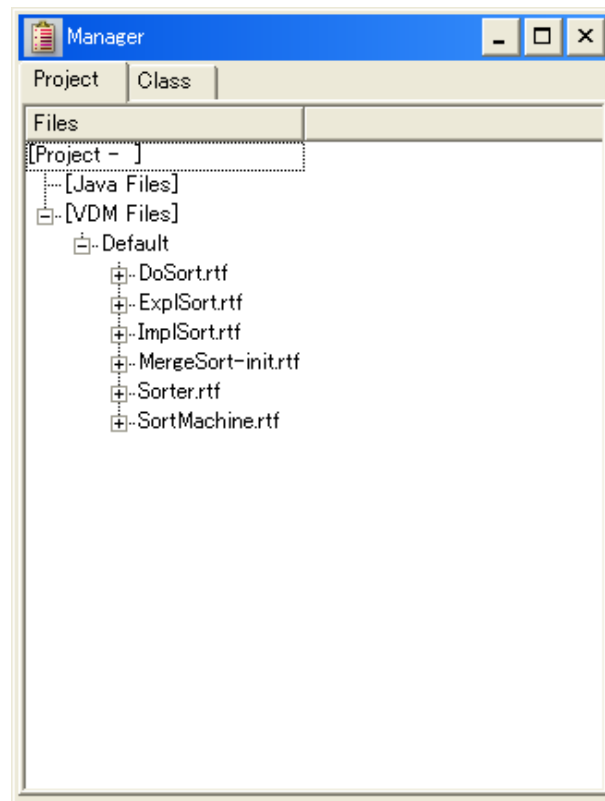



Figure 4: Main Window After Addition of Files

3.6 Syntax Checking your VDM Specification

Having configured your project you now need to check whether all the classes obey the syntax rules of VDM++. The syntax checker checks whether the syntax of your specification is correct. Note that when you change a source file you must syntax check it again before the other tools will be aware of the changes you have made.

3.6.1 Parsing the specification

Select the files for syntax checking by marking the six `.rtf` files in the **Project View** of the **Manager**, then press the  (Syntax Check) button on the (Actions) toolbar to invoke the syntax checker. (Selecting the containing level “Default” folder and applying the syntax check operation to that has the same effect – this applies the operation to each of the files in the folder.) Notice that at this

point the Log Window opens automatically (if it is not already open) and displays the message “Parsing “..../DoSort.rtf” ...” etc.. If syntax errors are discovered the Error List is also automatically invoked and the Source Window is automatically opened. Our sorting example contains two deliberate syntax errors by way of illustration.

3.6.2 Correcting syntax errors

The Error List is shown in Figure 5. Its top pane shows a list of the places (file name, line number, column number) at which errors and warnings arose, while the bottom displays a more detailed explanation of the currently selected error. Initially, the first error in the list is selected automatically.

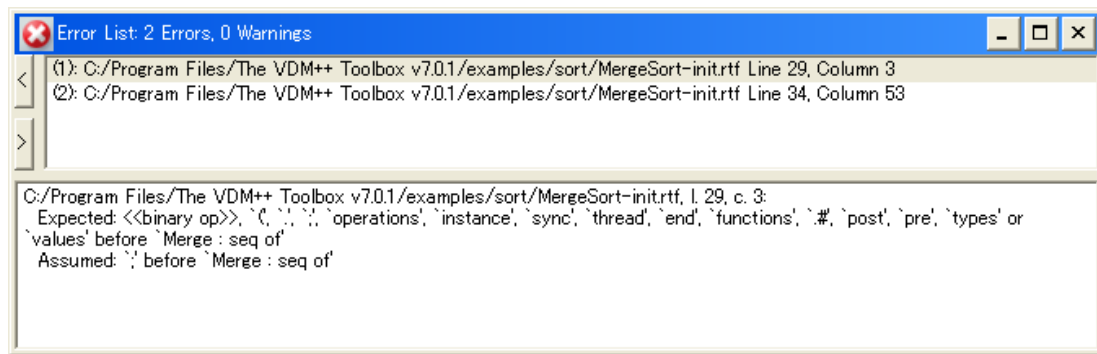


Figure 5: The Error List

The Source Window displays the part of the source specification in which the currently selected error was discovered, the actual position being marked by the window’s cursor. For the first syntax error, the Source Window appears as shown in Figure 6.

The first error message is as follows:

```
C:\vpphome\examples\sort\MergeSort-init.rtf, l. 29, c. 3:
Expected: <<binary op>>, '(', '.', ';', 'operations', 'instance',
          'sync', 'thread', 'end', 'functions', '.#', 'post'
          'pre', 'types' or 'values' before 'Merge : seq of'
Assumed: ';' before 'Merge : seq of'
```

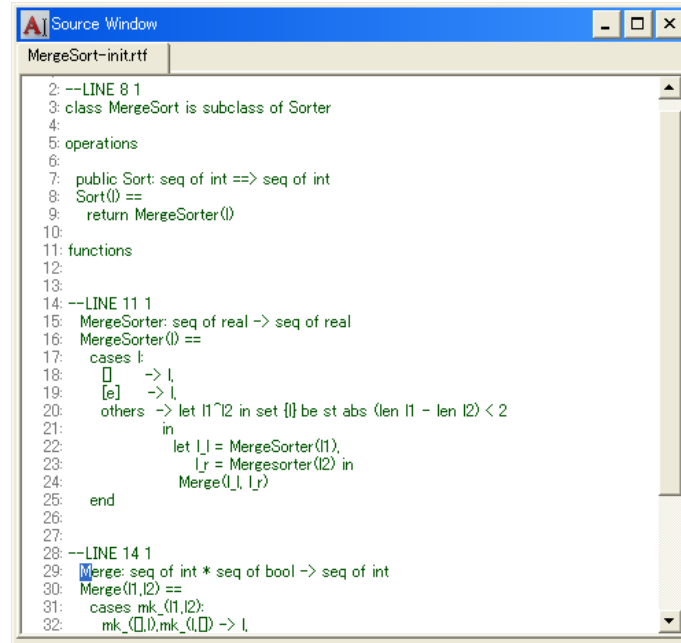




Figure 6: The Source Window for the First Error

Messages of this form indicate the text which was expected but not found at the error point. The syntax checker reports the error and makes an assumption about what should have been at the error point in order to allow it to recover and carry on with the syntax check.

In this example, the error message tells us that a missing ‘;’ before the function **Merge** has been assumed in order to proceed with the syntax check, and you can see from the description of the VDM++ syntax in [4] that this assumption is correct – two function definitions must be separated by the delimiter ‘;’. The error can therefore be fixed by using the file editor to add the character ‘;’ to the end of the definition of the function **MergeSorter**.

(Note that the source file is not changed by the syntax checker when it “assumes” something: corrections to the source text should be done manually by the user.)

You can correct the syntax errors by invoking your preferred editor (see Appendix C) directly from the Toolbox interface. Select the file **MergeSort-init.rtf** in the main window and press the **External Editor** button () on the (File Operations) toolbar. Note that if more than one file is selected when you invoke the **External Editor** in this way you actually get one **External Editor** for each of the selected files.


You can get to the next reported error by pressing the  button which appears to the left of the list of errors or by selecting the error notifier directly in the top pane of the **Error List**. Here the explanation is:

C:\vpphome\examples\sort\MergeSort-init.rtf, l. 34, c. 53:

Expected: <<binary op>>, '(', ')', ',', '.', '[', '~', '.#',
 ''' or ', ... ,' before 'l1 , l2)'


Assumed: '*' before 'l1 , l2)'




This tells us that there is a syntax error before l1, l2 and that a multiplication symbol '*' was assumed between 'tail' and 'l1' in order to recover from the syntax error (the 'tail' symbol has been read as the name of an identifier). In this case the assumption is incorrect and the error is in fact that the 'tl' operation which returns the tail of a sequence has been wrongly written as 'tail'. Make this correction using the file editor.

When you have corrected the syntax errors and saved the file, you must re-run the syntax checker to check your corrections were right. This time the file should be syntactically correct, and if you switch to the **VDM View** in the (Class View of the) **Manager** you will see that the status of each of the six classes (DoSort, ExplSort, ImplSort, MergeSort, Sorter, SortMachine) in our specification is now marked with the symbol  in the syntax column indicating that it is syntactically correct. (See Figure 7.) Note that the blanks in the other columns mean that no attempt has yet been made to type check, code generate or pretty print the specification.

Now that the syntax checking has been completed successfully, the files can be selected for further processing directly in the **VDM View**.

3.7 Type Checking your VDM Specification

Once a specification has passed the syntax check, the type checker can be applied. This is invoked by pressing the  (Type Check) button on the (Actions) toolbar.

Select all six classes in the **VDM View** and run the type checker. After type checking, the Toolbox updates the status information in this view to indicate, using the symbols  and  (this is the first symbol  with a red line through it) respectively, whether the type check succeeded or failed for each class.

In this example it in fact failed for the **MergeSort** class, which generated three

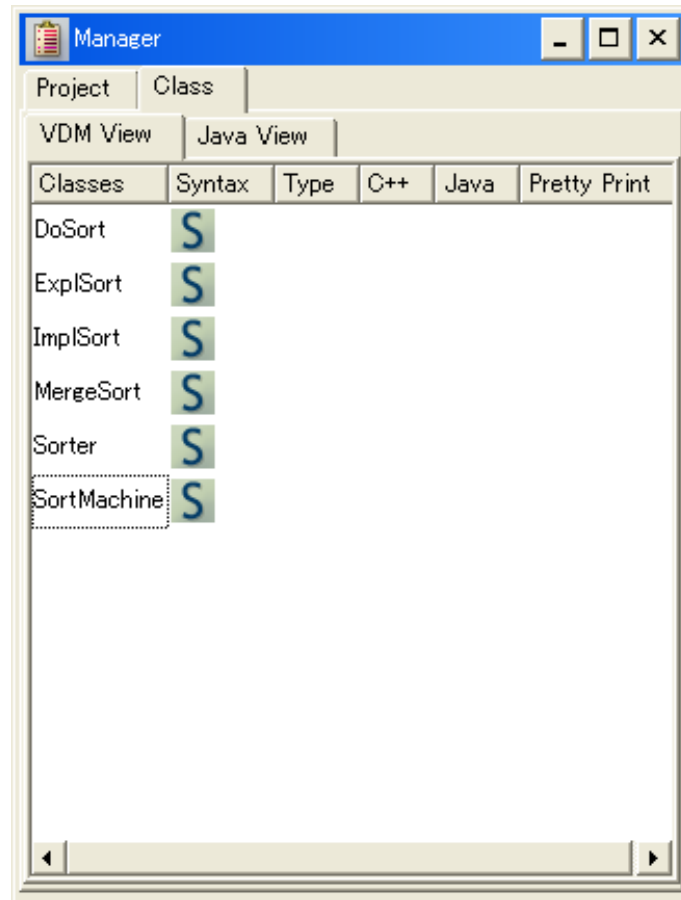


Figure 7: The VDM View

type errors and one warning. The errors⁶ are displayed in the **Error List** as before.

The first error, which is shown in Figure 8, is caused by the lower-case **s** in the function name **Mergesorter** (see the corresponding **Source Window** shown in Figure 9): this function name should be **MergeSorter**.

The second error, which is shown in Figure 10, tells us that we tried to apply the '**<=**' operator with a right-hand argument (Rhs) which does not belong to a numeric type. More specifically, the actual argument (denoted by the keyword **act:** in the error message) is of type **bool** while the expected argument (denoted by the keyword **exp:**) is a real number (**real** is the most general numeric type). Information like this can be valuable when trying to determine the cause of an

⁶The format of type errors is described in more detail in the reference part of this manual. See Section 4.4.

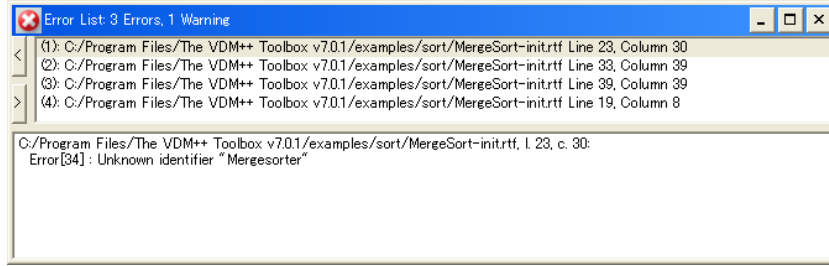


Figure 8: First error reported when type checking

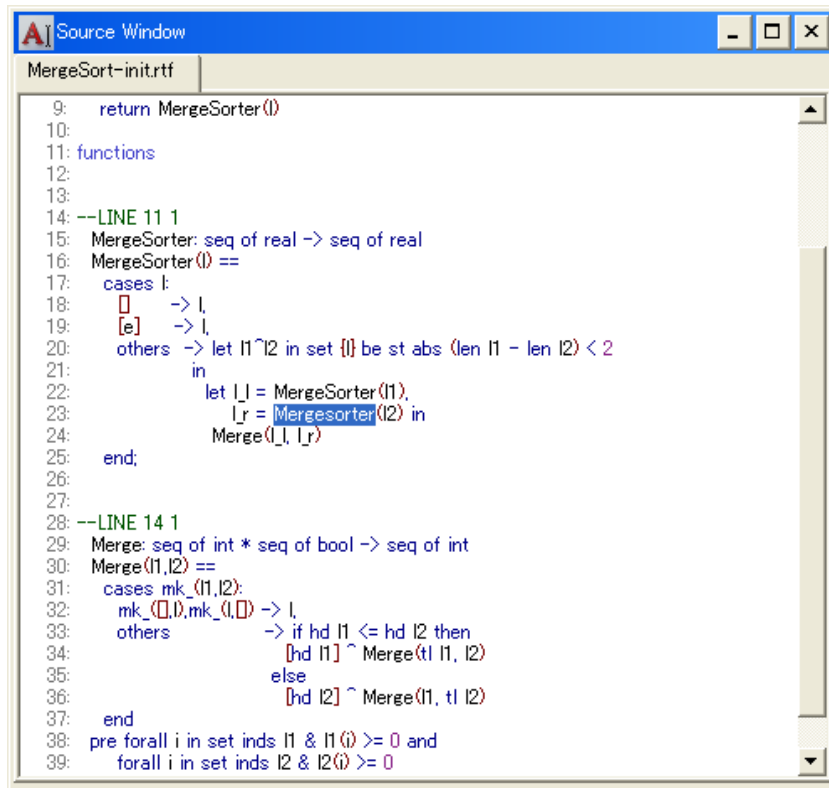


Figure 9: The Source Window for the Type Errors

error.

This mistake is in fact caused by the `seq of bool` in the signature of the `Merge` function, which should be `seq of int`. The same mistake also caused the third error, which is similar to the second, and this disappears when the second error is removed. So just correct the first two errors and syntax and type check the

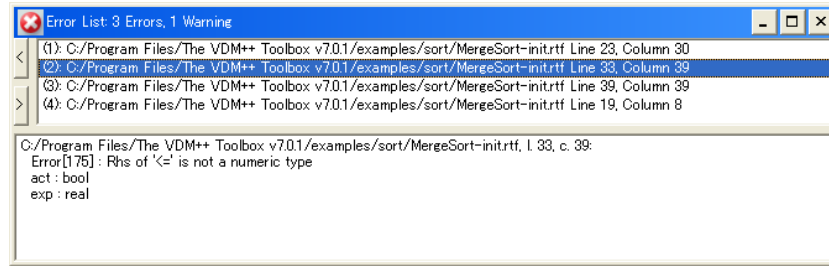


Figure 10: Second error reported when type checking

specification again.

Notice how the status information in the main window is updated during this process. First, when the source file is edited the symbol **S** which indicated in the VDM View that the file was syntactically correct is replaced by the symbol **S** indicating that there is an inconsistency between the version currently in the Toolbox and the version on the file system. The file must be syntax checked again before proceeding. Second, after the syntax check and type check are re-run, the symbols **S** and **T** are shown in the respective status fields to indicate that both operations were successful.

Note also that the type check operation is considered to be successful even though the type checker returns a warning. This is because warnings generally represent redundancy in a specification rather than actual errors, for example that a particular Boolean expression will always evaluate to false or that a particular parameter or local variable is never used in the body of a function or operation. Of course, such redundancy can actually be the result of an error – the expression or statement which raises the warning may have been mis-typed – so it is useful to check the warnings to make sure that this is not the case. In our example, the warning tells us that the local variable ‘e’ which is introduced in the second pattern in the cases statement in the function **MergeSort** is never used. This is in fact not an error and the specification is correct as it stands, but we could remove the warning if we wanted to by replacing the ‘e’ with the “don’t-care” pattern ‘-’.

Although our specification has now passed the type checking operation this does not mean that it is guaranteed to be correct and there may still be some errors (just as there may be run-time errors such as division by zero in a program even though that program has passed the syntax and type checks of the compiler for the appropriate programming language). In order to help to identify potential sources of these “run-time” errors in the model, the type checker has an option which

causes it to report an error at all points in the specification which are potential sources of run-time errors. Then, if one can convince oneself that the potential errors reported cannot occur, no run-time errors will appear. More information about this can be found in the reference part of this manual in Section 4.4.


3.8 Validating your Specification

Specifications are developed for a purpose: usually in order to gain a better understanding of the desired behaviour of a proposed computing system, or in order to check that some design has desired properties such as safety, or in order to serve as the basis for subsequent detailed design or coding. Whatever its purpose, it is not sufficient for a specification merely to be syntax- and type-correct – it must also faithfully express the behaviour of the system being modelled, albeit at an abstract level.

Validation is the process of increasing confidence that a formal specification accurately reflects the informally expressed requirements for the system which is being modelled. A wide range of validation techniques are available when the specification is given in a formal specification language: specifications may be inspected and they may be tested; it is even possible to conduct highly rigorous proofs that specifications exhibit desired properties. The Toolbox provides support for validation through animation and testing using the interpreter and the debugger – executing parts of the specification on chosen input values – and through the generation of integrity properties. This section shows you how the interpreter, the debugger and the integrity examiner can be used to check your specification and improve its quality.

3.8.1 Evaluating expressions using the interpreter

The interpreter allows you to evaluate and debug expressions and statements. These can be arbitrarily complex, including application of functions and operations and use of variables defined in the scope of the specifications read into the Toolbox. The debugger allows you to set breakpoints, step through the evaluation, and inspect variables.

The Interpreter Window, shown in Figure 11, is opened by pressing the  (Interpreter) button on the (Window Operations) toolbar. The Interpreter toolbar is opened at the same time if it is not already open.

The top two panes of the tool are respectively the Response and Dialog panes:

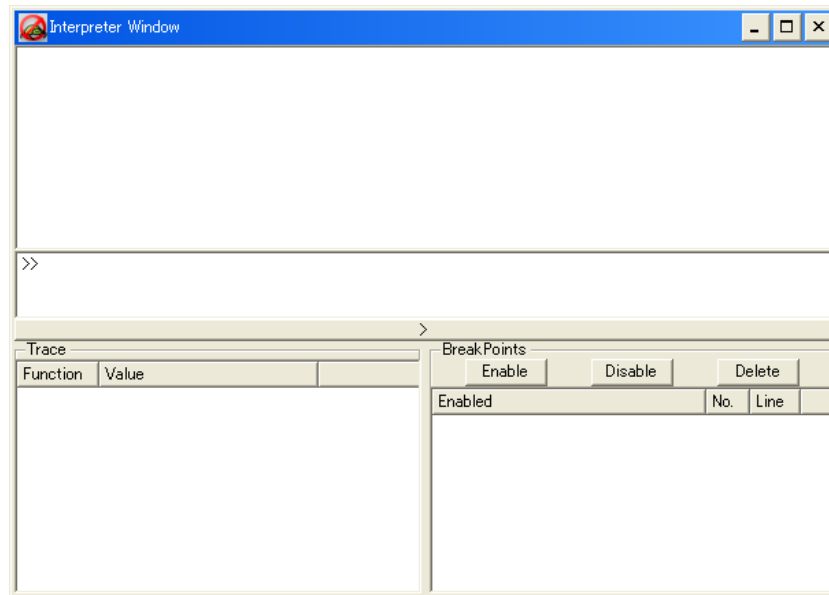



Figure 11: The Interpreter Window

you can give commands directly to the interpreter in the **Dialog** pane and you receive output from the interpreter in the **Response** pane. To evaluate a VDM++ expression, you type it directly on the command line in the **Dialog** pane.

Type the following expression into the **Dialog** pane:

```
print { a | a in set {1,...,10} & a mod 2 = 0 }
```

It is also possible to refer to the VDM++ constructs which have been read in from the specification files, but before doing this you must first initialise the interpreter by pressing the  (**Init**) button. During initialisation, constants are evaluated and instance variables are initialised. After initialisation you can refer to any of the functions, operations, instance variables, values, types etc. which are defined in the classes in the specification.

then press **RETURN**. The answer is a set of even numbers. The expression you evaluated was a set comprehension, a value construction which is explained further in [\[4\]](#).

In order to see which functions you can call from the class **MergeSort**, type **functions MergeSort** at the prompt in the **Dialog** pane. This displays the list of available functions in class **MergeSort**. From this list it can be seen that it is

possible to call the precondition function for functions which have a precondition attached to them, like the function `pre_Merge`.

Application of functions and operations and inspection of instance variables and values can only be performed through objects. Objects can be created so that they are available for subsequent use in the interpreter.

The following two commands, when used in the **Dialog** pane, will create an object of class `MergeSort` named `ms` then call the operation `Sort` on `ms` with the sequence `[3.1415, -56, 34-12, 0]` and display the result:

```
create ms := new MergeSort()
print ms.Sort([ 3.1415, -56, 34-12, 0 ])
```

Alternatively, objects can be created local to the evaluation of a statement in the interpreter. For example the call to `MergeSort.Sort` could also be performed by the following command where the `ms` object from the previous example is not given any name at all:

```
print new MergeSort().Sort([ 3.1415, -56, 34-12, 0 ])
```

In this example an object is created local to the evaluation, i.e. it only exists throughout the evaluation of the `Sort` operation.

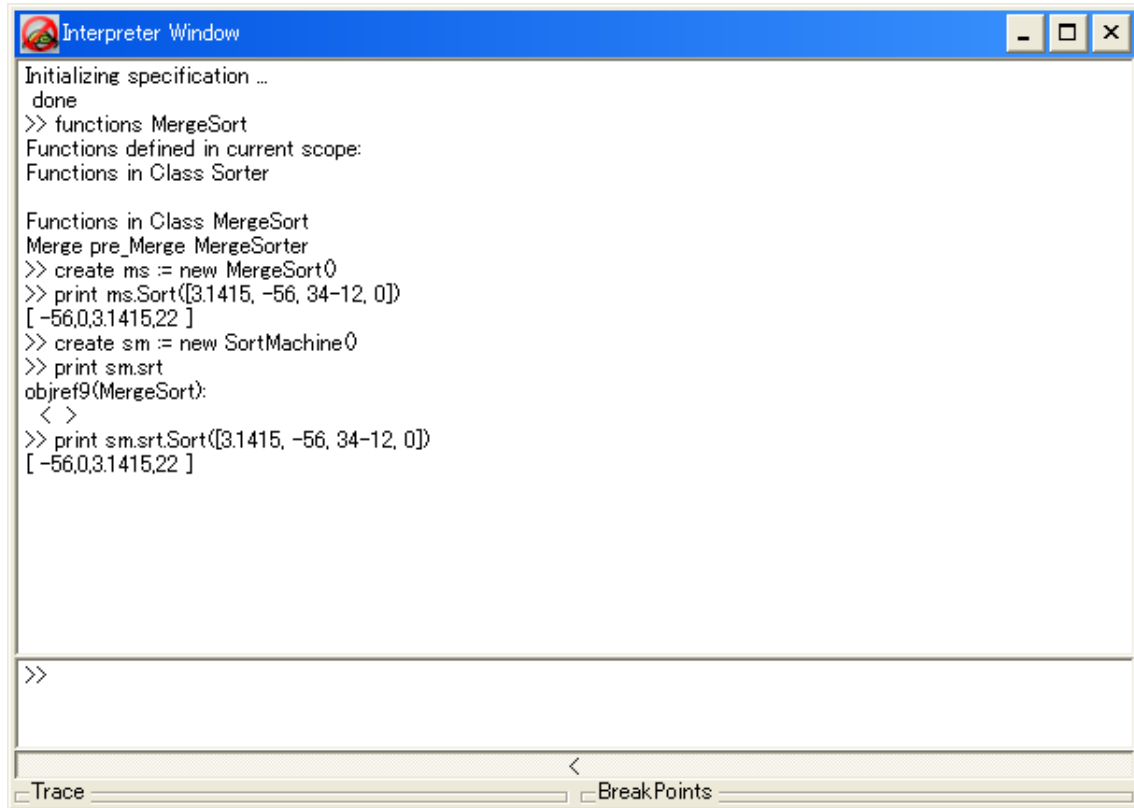
In order to look up an instance variable, first create an object of the class `SortMachine`:

```
create sm := new SortMachine()
```

The class `SortMachine` has an instance variable `srt` that is a reference to a `Sorter` object. Initially `srt` points to a `MergeSort` object. Try first to see the value of `srt` and then call the `Sort` operation on the object it points to:

```
print sm.srt
print sm.srt.Sort([ 3.1415, -56, 34-12, 0 ])
```

Figure [12](#) shows the results of all the above evaluations.



```

Interpreter Window
Initializing specification ...
done
>> functions MergeSort
Functions defined in current scope:
Functions in Class Sorter

Functions in Class MergeSort
Merge pre_Merge MergeSorter
>> create ms := new MergeSort()
>> print ms.Sort([3.1415, -56, 34-12, 0])
[-56,0,3.1415,22 ]
>> create sm := new SortMachine()
>> print sm.srt
objref9(MergeSort):
< >
>> print sm.srt.Sort([3.1415, -56, 34-12, 0])
[-56,0,3.1415,22 ]

>>

Trace BreakPoints

```

Figure 12: Evaluation of Expressions

Note that not all VDM++ constructs are executable and unexecutable constructs cannot be evaluated using the interpreter. For example if you try to call the implicitly defined function `ImplSort` with a sequence of numbers, the interpreter will return an error saying that it encountered a non-executable construct during evaluation. See Figure 12.

3.8.2 Setting breakpoints

Breakpoints cause the interpreter to break execution when evaluating functions or operations.

Set a breakpoint in the function `MergeSorter` in the class `MergeSort` by typing `break MergeSort 'MergeSorter`.

(When setting a breakpoint the name of the function or operation must be qualified with the name of the class in which it is defined.) The location of the breakpoint will now appear in the **BreakPoints** pane at the bottom right of the **Interpreter Window** together with the number allocated to the breakpoint (1 in this case since this is the first breakpoint we have set) and the symbol ☒ which indicates that the breakpoint is enabled. You can now use the **debug** command to start the evaluation instead of the **print** command used before. The only difference between the two commands is that **debug** forces the interpreter to stop at breakpoints whereas **print** ignores breakpoints.

```
debug new MergeSort().Sort([ 3.1415, -56, 34-12, 0 ])
```


will cause the interpreter to stop at the breakpoint when it enters the function **Sort**. At the same time, the source file containing the specification of the function **Sort** is displayed in the **Source Window** and the current point of evaluation (at the moment this is the location of the breakpoint, i.e. the beginning of the function **Sort**) is indicated by the cursor. In addition, the **Trace** pane, which is situated at the bottom left of the **Interpreter Window**, shows the function call stack.

You can now inspect the values of the parameters of the **Sort** function, either by printing them using **print** (e.g. by typing **print 1** in the **Dialog** pane) or by clicking the left mouse button on the ‘...’ adjacent to the function name in the **Trace** pane at the bottom left of the **Interpreter Window**. Clicking the left mouse button on the parameters which are revealed will replace them with the ‘...’ again.



You can also set breakpoints by selecting the desired position directly in the appropriate source file. In addition, breakpoints need not be at the start of a function/operation but can be at any position within its body.


If the source file is not an RTF file you can set breakpoints by double-clicking the left or middle mouse button on the desired position in the file in the **Source Window**. If you are using an RTF source file you must position the cursor at the appropriate position in the file in Microsoft Word, then press **Control-Alt-spacebar** to set a breakpoint.

You can set breakpoints at any time during debugging, so now use the source file to set a new breakpoint inside the function **Merge** in the class **MergeSort** as described above.

Return to the interpreter and press the  (**Continue**) button on the (**Interpreter**) toolbar. This causes the execution to carry on to the next breakpoint. In fact

because of the recursive call of `Sort` the interpreter will stop at the same breakpoint again, so press the **Continue** button repeatedly until the execution stops inside the `Merge` function.

As the execution proceeds, the various function/operation calls are logged in the **Trace** pane, and you can use this function call stack to navigate through the steps of the execution so far. Press the  (Up) button a couple of times to see how the position in the function trace context can be changed. The  (Down) button can be used to move back down the trace again.

You can also step through the execution expression by expression by pressing the  (Single Step) button. Press this a few times and see how the cursor in the **Source Window** moves to mark the changes in the current point of evaluation. You now have access not only to the parameters of the function but also to all the variables (including local variables) that are in scope at the current point of evaluation, and you can inspect their values using, for example, the `print` command.

This debugging is shown in Figure 13.

You can delete breakpoints, also at any time during the debugging. Try this by typing `delete 1` (i.e. delete breakpoint number 1) in the **Dialog** pane of the **Interpreter Window**. Selecting the breakpoint in the **BreakPoints** pane of the **Interpreter Window** and pressing the **Delete** button at the top of that pane has the same effect.

The other two buttons at the top of the **BreakPoints** pane are for enabling and disabling breakpoints. Set the breakpoint in the function `MergeSort` 'MergeSorter' again, then select it in the **BreakPoints** pane and press the **Disable** button. Note how the symbol ☒ is replaced by the symbol ☐ to indicate that the breakpoint is disabled. Pressing the **Enable** button will re-enable the breakpoint and the symbol will change back to ☒ to confirm this.

3.8.3 Dynamic type checking

Although the type checker reported no errors in our specification, there may still be type errors because in general it is not possible to find all type errors by a simple static analysis of type information (i.e. an analysis based only on the types declared in the signatures of functions and operations). Thus, for example, if a function is declared as taking an integer (of type `int`) as its argument but is applied to an expression which evaluates to a real number (of type `real`) this will

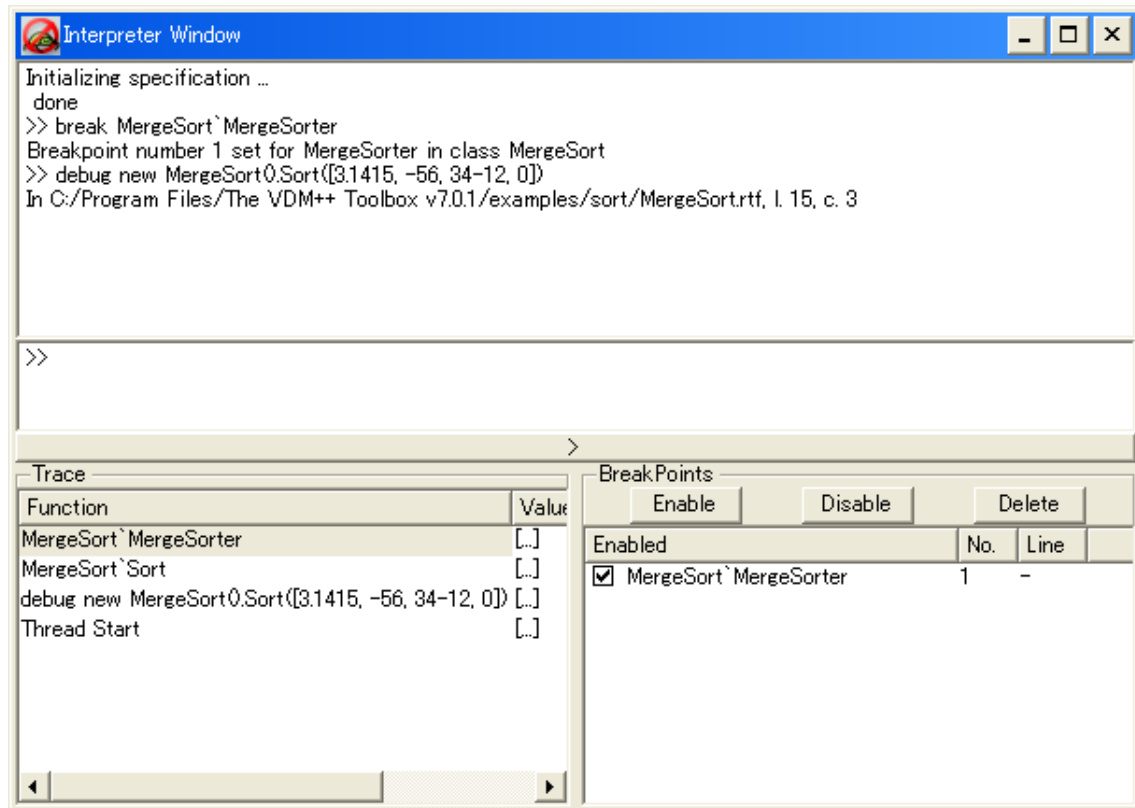



Figure 13: Debugging a Specification

not raise a static type error because `int` is a subtype of `real` so the application might be correct provided the function is called at run-time with parameters which are actually integer reals.

In order to help discover this kind of type error at the specification level, the interpreter can be configured to perform dynamic type checking during evaluation. This option is enabled through the **Interpreter** pane of the **Project Options** window, which is displayed by pressing the  (Project Options) button on the (Project Operations) toolbar. This is shown in Figure 14.

Enabling dynamic type checking causes the interpreter to check actual types during an evaluation.

To see an example of this, enable dynamic type checking by selecting it in the **Interpreter** pane of the **Project Options** window and pressing the **OK** button to accept the new options. Then evaluate the expression

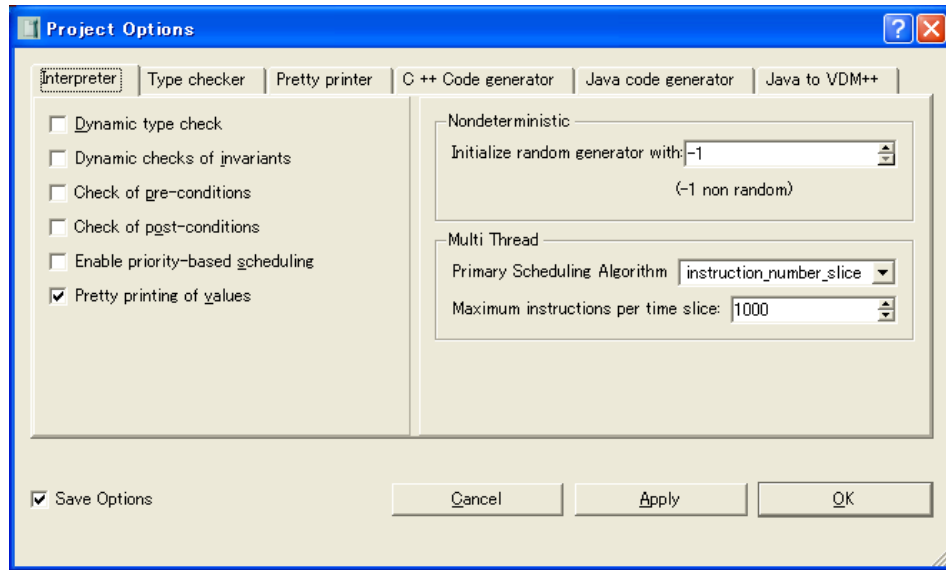


Figure 14: Setting Interpreter Options

```
debug new MergeSort().Sort([ 3.1415, -56, 34-12, 0 ])
```

again in the Dialog pane. This time the interpreter reports a dynamic type error. (If you have not deleted or disabled all breakpoints you will need to step through the specification to see this.) This is because according to its signature the function `MergeSort.Merge` expects sequences of integers as its parameters whereas the actual parameters contain the value `3.1415` which is not an integer but a real number. This dynamic type error thus reveals a possible error in the specification – the top-level function `MergeSort` in the `MergeSorter` class can accept sequences of real numbers as its parameters but it calls the function `Merge` in the same class which is only defined for lists of integers.

In a similar way, the interpreter can be configured to dynamically check that invariants on types and preconditions and postconditions of functions and operations are respected (i.e. evaluate to `true`) during evaluation. These options are also enabled through the **Interpreter** pane of the **Project Options** window shown in Figure 14.

As an example, return to the **Interpreter** pane of the **Project Options** window and enable the option **Check of pre-conditions**. Now evaluate the previous expression again but this time omitting the number `3.1415` from the list. Now the interpreter reports a precondition violation as shown in Figure 15. This is because the precondition of the function `MergeSort.Merge` requires that all the input values

should be non-negative.

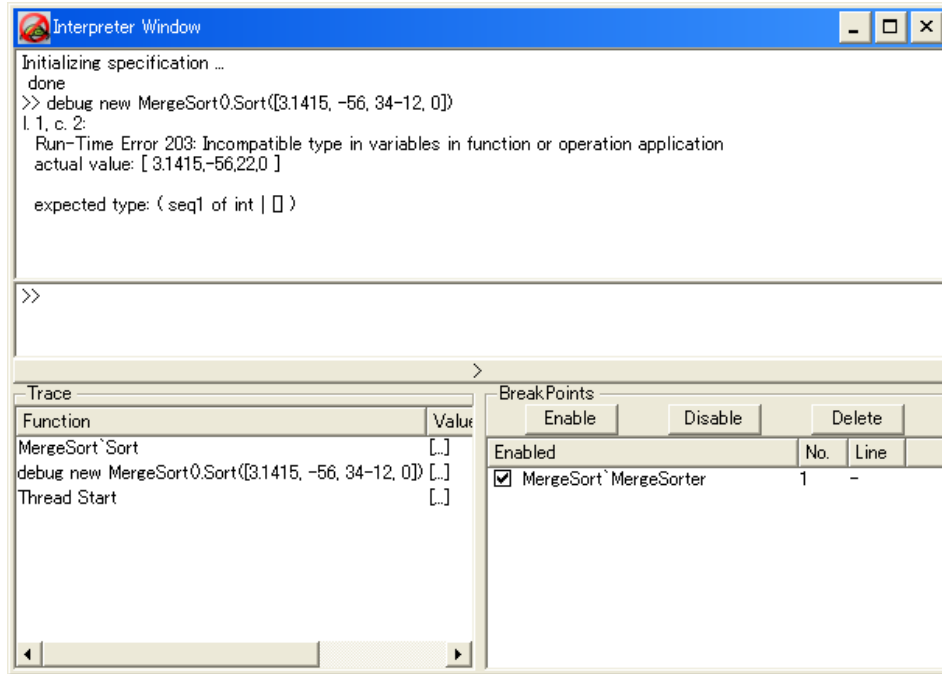


Figure 15: Dynamic Type Checking Error


3.8.4 Checking Integrity Properties

Dynamic checking of types, invariants, preconditions and postconditions as described above is basically one form of testing – it checks for run-time errors for some *specific* input values. The integrity examiner offers a more general way of investigating possible run-time errors, though it is perhaps less intuitive for people who are more familiar with programming than with mathematics.

The integrity examiner analyses the specification looking for places where run-time errors could potentially occur and generates a series of integrity properties which represent conditions under which no run-time errors should occur. These integrity properties are more general than dynamic checking because they are presented as VDM++ predicates that involve quantification over all possible values of the appropriate variables⁷, which means that if it can be demonstrated that an integrity property is true there will not be run-time errors associated

⁷In some cases the full context is not shown explicitly and the scope of some variables has to be determined by inspection of the specification.

with that integrity check whatever the values of the variables involved (dynamic checking of course only checks that there are no run-time errors for the particular values of the variables chosen). Of course if an integrity property can instead be shown to be false this would point to there being a potential problem with the corresponding part of the specification.

To see how the integrity examiner works in practice, select the `ExplSort` class then invoke the integrity examiner for this class by pressing the  (Generate Integrity Properties) button on the (Actions) toolbar. The Integrity Properties Window then opens and displays the integrity properties generated. This is shown in Figure 16.

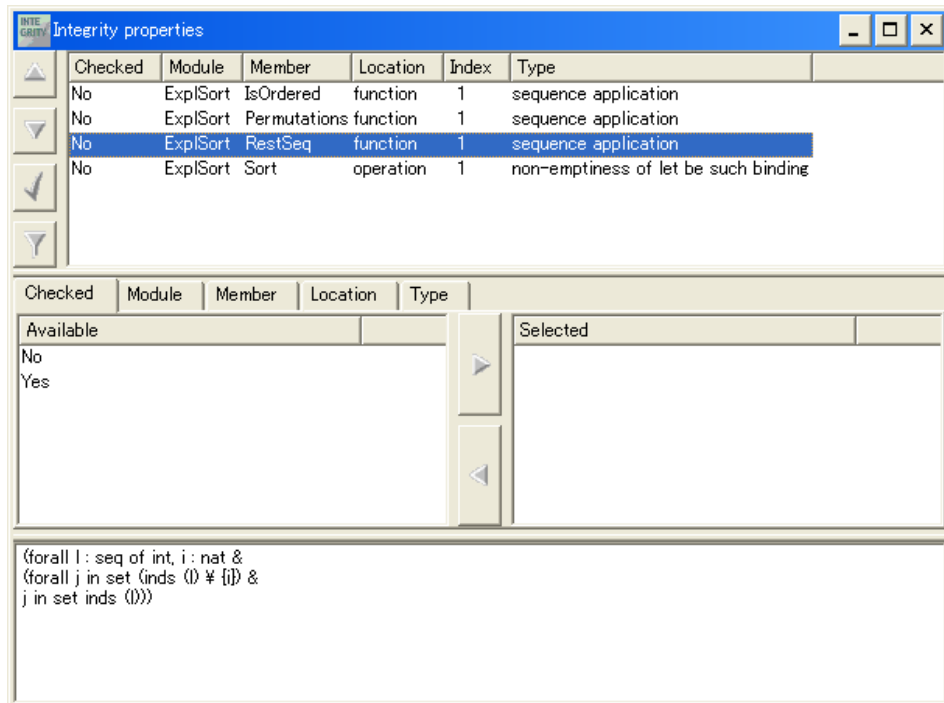


Figure 16: The Integrity Properties Window

The top pane of the Integrity Properties Window shows a list of the integrity properties together with information about their status (the **Checked** column), their position in the specification (the **Module**, **Member** and **Location** columns) and their type (the **Type** column). The numbers in the **Index** column simply serve to distinguish different integrity properties which have the same position. As can be seen from this example, even a small specification can generate many integrity properties – in fact thirty different types of integrity properties are checked in all – so in a large specification it is useful to be able to filter these.

The middle two panes of the **Integrity Properties Window** offer various filtering methods (see Section 4.6 for details). Finally, if a particular integrity property is selected in the top pane of the window the corresponding VDM++ predicate is displayed in the bottom pane of the window, and at the same time the cursor in the **Source Window** indicates the exact point in the specification to which the selected integrity property relates. Each integrity property can thus be inspected in order to try to determine whether or not it is true.


Select the first (i.e. index number 1) integrity property relating to the function `isOrdered`. This has the form:

```
(forall l : seq of int &
  (forall i,j in set inds (l) &
    i > j =>
      i in set inds (l)))
```

and, as can be seen from the position of the cursor in the **Source Window**, corresponds to the condition that the sequence application `l(i)` in the expression

```
forall i,j in set inds l & i > j => l(i) >= l(j)
```

must be well-defined, i.e. the value `i` must always belong to the indices of the sequence `l`.

In this particular case it is in fact easy to see that the integrity property is true – the second quantification in the predicate directly tells us that both `i` and `j` belong to the indices of `l`, and whether or not `i` is bigger than `j` (the third line of the predicate) is irrelevant. The property can therefore be marked as having been checked, which is done by pressing the  (**Toggle Status**) button to the left of the top pane of the **Integrity Properties Window**.

Now look at the other three integrity properties related to sequence application. It is also easy to see that these are true: the one relating to `isOrdered` is exactly analogous to the one discussed above except that it relates to the sequence application `l(j)` rather than to `l(i)`, so the same argument applies; the one relating to `Permutations` is immediately true because the second quantification gives exactly the result required (`i in set inds (l)`); and in the case of `RestSeq` the third quantification tells us that `j` belongs to the indices of `l` with `i` removed which means that `j` must belong to the indices of `l`. These three integrity properties can therefore be selected and marked as checked in the same way.

In cases such as these, the integrity properties could in fact be verified automatically by a mechanical checker. However, this is not always possible and in the more complicated cases the reasoning process needs to be steered by a human even though the actual reasoning can be mechanised.

One such more complicated property is the one relating to the **Sort** operation, which basically states that there must be at least one value **r** which satisfies the predicate in the implicit **let** statement otherwise the specification does not make sense⁸. It is not so easy to see that this property is true because it involves three user-defined functions – **Permutations**, **isOrdered**, and **RestSeq** which is used in the definition of **Permutations** – and in addition **Permutations** is defined recursively. However, it is easy to see that the integrity property is true *provided* the functions **Permutations** and **isOrdered** are defined correctly – clearly it is possible to sort any given sequence of numbers, so we just need to be sure that the set of sequences returned by the function **Permutations** comprises all possible permutations of the input sequence and that the function **isOrdered** defines ordered sequences of numbers correctly.

Look at the definition of the function **isOrdered** in the **Source Window**. It is relatively easy to see that this is correct – its defining predicate states directly that, given any two positions in the sequence, the number at the later position cannot be smaller than the number at the earlier position, and this clearly means that the elements must be in (ascending) order.

Now look at the function **Permutations**. The first branch of the cases expression is easy to deal with – there is only one possible permutation of the empty sequence and sequences with only one element, namely the sequence itself. For the **others** branch, we first need to look at the function **RestSeq**. It is fairly easy to see that this simply removes the element at a given position from a given sequence. In the **others** branch of the function **Permutations**, therefore, we are constructing permutations by choosing an arbitrary element from the original sequence as the first element of the permutation and concatenating all possible permutations of the remaining elements of the original sequence onto this. This therefore gives us all possible permutations, so the integrity property is satisfied.

Looking now at the remaining two integrity properties, both relating to the function **RestSeq**, it is easy to see that the one of type **Postcondition**, which requires that the explicit result of the function satisfies the postcondition if the precondition is satisfied, is valid – the function removes one element from the sequence so the length of the sequence is reduced by one and the elements of the sequence are

⁸There is an implicit quantification here over the variable **l** which, according to the specification, is an arbitrary sequence of integers.

either unchanged (in the case when the element removed occurs more than once in the sequence) or smaller. However, the property of type **Invariant** states that every natural number is different from zero, and this is of course false.

Looking at the specification of **RestSeq** in the **Source Window**, you can see that the property is generated by the precondition of the function:

```
i in set inds l
```

In fact it arises because the indices of a sequence is a set of positive natural numbers (i.e. is of type **set of nat1**) so that if **i** is not of type **nat1** the precondition will automatically be false. This indicates that the **nat** in the signature of the function should be changed to **nat1**. If this is done, the new integrity property will be

```
(forall l : seq of int, i : nat1 &  
i <> 0)
```

and this is of course true.

The integrity properties for the other classes can be dealt with in a similar way.

3.8.5 Multi-threaded models

VDM++ supports multiple threads within a model, and this feature of the language is also supported by the interpreter which allows you to insert breakpoints within particular threads and to step through threads. It also allows selection of a particular thread to step through. The scheduling algorithm used by the interpreter may be selected from a variety built into the Toolbox.

3.9 Introducing Systematic Testing

As part of its support for validation, the Toolbox provides a facility for testing VDM++ specifications, including test coverage measurement. Test coverage measurement helps you to see how well a given test suite covers the specification. This is done by collecting information in a special test coverage file about which statements and expressions are evaluated during the execution of the test suite.

There are three steps involved in producing a test coverage report:

1. Prepare a *test coverage file*. This file contains information about the specification's structure.
2. Test the specification by making the interpreter execute calls to the constructs in the specification. This process updates the test coverage information in the test coverage file.
3. Pretty print the test coverage report: the pretty printer takes the specification and test coverage files and produces a nicely typeset version of the specification with test coverage information included. We will return to this part below in Section 3.10.

```

Interpreter Window
Initializing specification ...
done
>> tcov reset
>> p new DoSort0.Sort([-12, 5, 45])
[-12,5,45]
>> tcov write vdm.tc
>> rtinfo vdm.tc
100% 1 DoSort`Sort
100% 4 DoSort`DoSorting
62% 3 DoSort`InsertSorted
0% 0 Sorter`Sort
0% 0 ExplSort`Sort
0% 0 ExplSort`RestSeq
0% 0 ExplSort`IsOrdered
0% 0 ExplSort`Permutations
0% 0 ImplSort`Sort
0% 0 ImplSort`IsOrdered
0% 0 ImplSort`ImplSorter
0% 0 ImplSort`IsPermutation
0% 0 MergeSort`Sort
0% 0 MergeSort`Merge
0% 0 MergeSort`MergeSorter
0% 0 SortMachine`SetSort
0% 0 SortMachine`GoSorting
0% 0 SortMachine`SetAndSort

Total Coverage: 13%

>> |

```

Figure 17: Collecting Test Coverage Information

This process is illustrated in Figure 17. First the `tcov reset` is issued to reset the test coverage file so that it has no information about any prior testing carried out for the given specification. Then the `print` command is used to evaluate different constructs from the specification. The command `tcov write` then saves all the test coverage information generated since the last `tcov reset` command to the file `vdm.tc`. Finally, the command `rtinfo` displays a table summarising

the information in this test coverage file. This consists of a list of the various functions and operations in the specification together, each annotated with the number of times that function/operation has been called during testing and the percentage of its specification which has been tested at least once.


Note that the command-line version of the VDM++ Toolbox (`vppde`) also has facilities to support the collection of test coverage information.

Naturally realistic testing would involve many more tests before the information is written to the test coverage file `vdm.tc` using the `tcov write` command. Indeed, for real projects you would generally set up an entire test environment where you make a small script file which automates this whole process. This can also compare the actual results of individual tests against expected results (it is necessary to use the `-O` option for this). Appendix E contains an example of such a script file for both Windows and Unix.

3.10 Pretty Printing

The pretty printer transforms a specification from its input format to a pretty printed version of the specification. Typically this pretty printed version is used for documentation purposes.

In order to see pretty printing at work, first go to the **Pretty Printer** pane of the **Project Options** window and enable one of the options to produce indexes (it does not matter which of the two options you choose when the RTF format is used) and also the test covering colouring option. You also need to copy the `vdm.tc` file you have just produced to the working directory of the Toolbox, which you can determine using the `pwd` which you can run in the **Dialog** pane of the interpreter⁹.

Select all six `.rtf` files in the **Project View** in the **Manager**, then press the  (**Pretty Print**) button on the (**Actions**) toolbar. In the **Log Window** you will see that this produces a `.rtf.rtf` file for each of the selected input files. Start Microsoft Word on the `dosort.rtf.rtf` file. Notice how all the VDM++ keywords have been converted to boldface type. The other parts of your specification have been typeset using the Word styles `VDM_COV` and `VDM_NCOV` which relate to the covered and non-covered parts respectively. The definition of these styles can be changed and unless you use a colour printer for your documents it is necessary to modify the definition of the `VDM_NCOV` style (e.g. by using grey for the non-covered parts).

⁹If the project you are working on has been saved then the directory in which it was saved will be the working directory.


Go to the bottom of the `dosort.rtf.rtf` file. Note how the text written in the `VDM_TC_TABLE` style has been replaced with a table showing the test coverage statistics. The three columns give respectively the name of the function/operation, the number of calls of that construct in the test coverage file, and the percentage coverage for it. The table looks like¹⁰:


name	#calls	coverage
DoSort	4	100%
ExplSort	0	0%
InsertSorted	3	62%
IsOrdered	0	0%
IsPermutation	0	0%
Merge	0	0%
MergeSort	0	0%
Permutations	0	0%
RestSeq	0	0%
total		15%

Finally, go to the end of the file and select the **Index and Tables ...** item from the **Insert** pull down menu inside Microsoft Word. Decide the layout you wish to use for the index overview of the definitions in the `dosort.rtf.rtf` file. Press **Ok** and see how an index of VDM definitions can be created automatically.

Using the alternative pretty printing mechanisms with \LaTeX is quite different but this is explained in the reference section of this manual (see Section 4.10).

3.11 Generating Code

If you have a license for the VDM++ to C++ Code Generator you can automatically have your specification translated into C++ code by pressing the  (**Generate C++**) button. See [7] for further information about the C++ Code Generator.

Similarly, if you have a license for the VDM++ to Java Code Generator you can automatically have your specification translated into Java code by pressing the  (**Generate Java**) button. See [8] for further information about the Java Code Generator.

¹⁰Note that this is quite similar to part of the information we saw directly inside the **Response** pane of the interpreter in the previous section.

3.12 The VDMTools API

All of the functionality of **VDMTools** is exposed to external programs via a Corba-compliant application programmers interface (API). Details of how to use this API may be found in [\[9\]](#).

3.13 Exiting VDMTools

When you wish to exit the Toolbox you should select the **Exit** item on the **Project** menu in the main window. If you exit without saving the project, a dialog window will appear asking if you want to save your project.

This completes the guided tour of the Toolbox. We hope that you now have a better understanding of the kind of functionality it can provide. Now you should be able to start using the Toolbox for your own VDM++ models. The remaining parts of this manual are a detailed reference guide providing more details about particular features.

4 The VDMTools Reference Manual

This section is structured into a number of subsections covering each of the tools in the Toolbox. For each tool, its use through each of the three interfaces (the graphical user interface, the Emacs interface and the command line interface) is described.

4.1 The Overall Graphical User Interface

The graphical user interface to the Toolbox is started by selecting it from the programs entry in the Windows setup under Windows or with the command `vppgde` on Unix platforms. This opens the main graphical user interface window, which is shown in Figure 18.

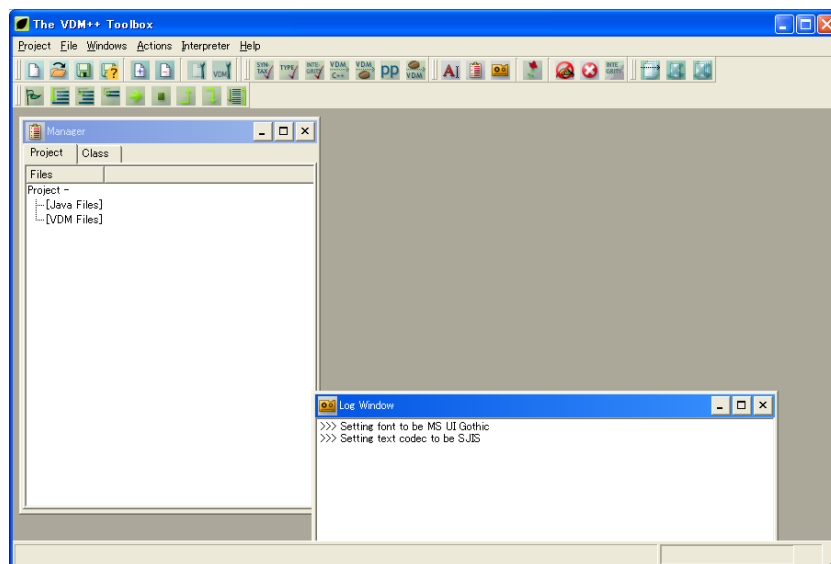


Figure 18: Graphical User Interface Startup

The top of this window consists of a menu line with six pull-down menus, below which are six toolbars¹¹ comprising buttons which offer the same actions as the menus¹². The bottom part of the window is used to display various subwindows which either present information about the status of the current project or offer


¹¹When the Toolbox is started, only three toolbars are displayed open, the other three being displayed in iconised form above them.

¹²Except that the function for exiting from the toolbox is only available on the Project menu.

interfaces to tools within the Toolbox. We describe each of the menus/toolbars and the available subwindows, grouped according to functionality, in the following subsections.












4.1.1 Project handling

A project consists of a collection of files that together form a VDM++ specification. Projects can be saved to and read from disk, which means that you do not need to configure the Toolbox with the individual files every time you wish to use it: you simply open the relevant project file. Projects are only available in the graphical user interface.

The **Manager**, which is opened/closed by pressing the  button on the **Window Operations** toolbar or by selecting the appropriate item from the **Windows** menu, displays the current status of the current project and is also the place where you select which subset of project files you want the various Toolbox operations to be applied to. It consists of two parts: the **Project View** and the **Class View**.

The **Project View** displays a tree representation of the contents of the project comprising the files in the project and (only after successfully syntax checking the file) the classes declared in each file. It is shown in Figure 19.

The **Class View** comprises both the **VDM View** and the **Java View**.

When VDM++ files have been successfully syntax checked the names of the classes defined in those files are listed in the **VDM View**. This view also displays the status of each of the individual classes in the project: the symbols , , , , and  in the appropriate columns indicate respectively that the class has been successfully syntax checked, type checked, translated to C++, translated to Java, and pretty printed; similarly, the corresponding symbols with a (red) line through them (, , , , and ) indicate that the particular action failed. Note that a blank in a column means that no attempt has yet been made to perform that particular action. Note also that if one of the files in the project is modified on the file system the symbol  is displayed in the **Syntax** column to indicate that there is an inconsistency between the version currently in the Toolbox and the version on the file system and that the file should be syntax checked again before proceeding.

The **Java View** is analogous to the **VDM View** except that it shows the names and the status of classes defined in Java files (again these must have been successfully syntax checked in order for anything more than the file name to appear). The

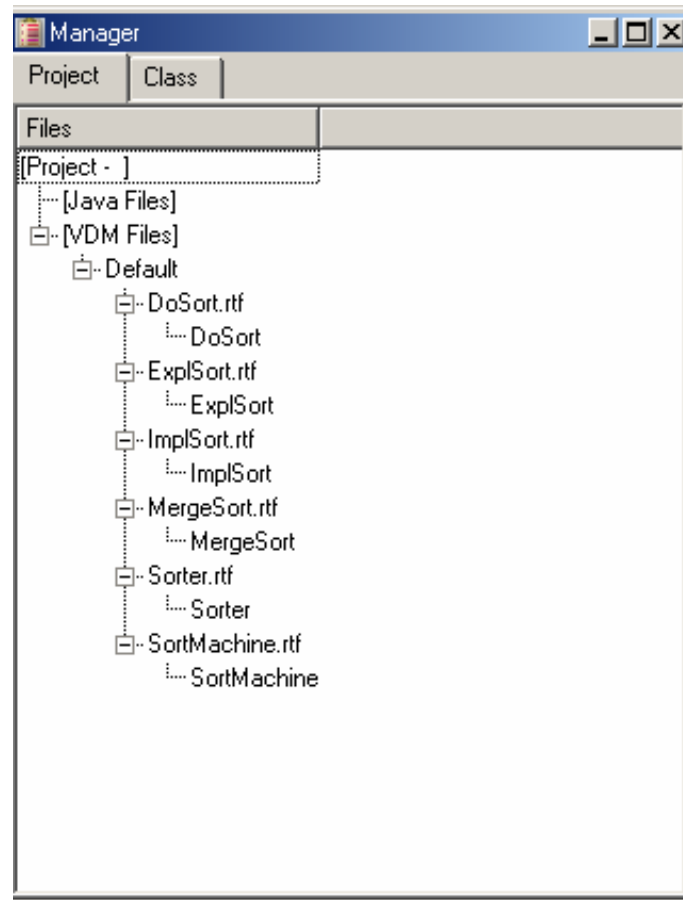


Figure 19: The Project View

symbols , , and are again shown in the appropriate column to indicate whether or not the class has been successfully syntax checked and type checked, and the symbols and in the third column denote respectively that the class has been successfully or unsuccessfully translated from Java to VDM++. Again a blank in a column means that no attempt has yet been made to perform that particular action, and if one of the files in the project is modified on the file system the symbol is displayed in the **syntax check** column to indicate that there is an inconsistency between the version currently in the Toolbox and the version on the file system and that the file should be syntax checked again before proceeding.

Various operations for manipulating projects, including opening and saving projects, adding files to and removing files from projects, and creating new projects, are available from the **Project** menu and the corresponding **Project Operations** toolbar,

which are shown in Figure 20.

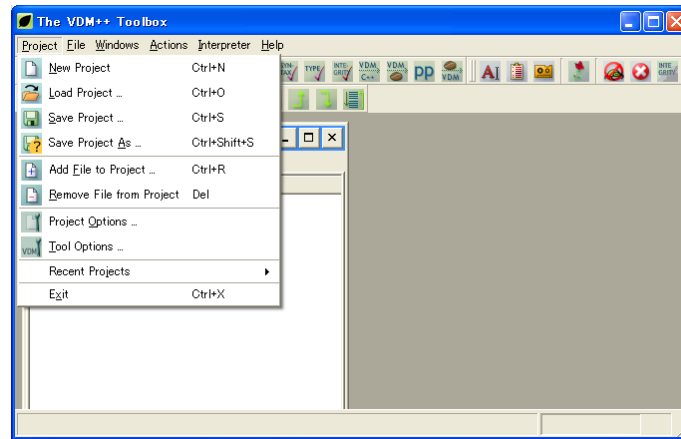





Figure 20: The Project Menu and Project Operations Toolbar

The same menu/toolbar also offer facilities for setting options relating to the Toolbox environment, for setting options for the various tools within the Toolbox, and for exiting the Toolbox (only available on the menu). In more detail, the available actions are as follows:

- New Project** (39

Project Options ... (

- interpreter (described in Section 4.5);
- type checker (described in Section 4.4);
- pretty printer (described in Section 4.7);
- C++ code generator (described in Section 4.8)
- Java code generator (described in Section 4.9);
- Java to VDM++ translator (described in [1]).

Tool Options ... (

Recent Projects: This opens a list of the recent projects that have been used on the computer.

Exit: Choose this item to leave the Toolbox. If you have not already saved your project the Toolbox will ask whether you wish to do so. Note that this action is not available on the toolbar.

4.1.2 Operations on specifications

The Toolbox offers a range of functions which can be applied to a specification: syntax checking; type checking; generating integrity properties; generating C++ or Java code; translation from Java to VDM++; and pretty printing. These are invoked through the **Actions** menu or the corresponding **Actions** toolbar, which are illustrated in Figure 21.

Each action is applied to every file/class which is currently selected in the **Manager**, though the actions are to a certain extent interdependent so that some of them can only be carried out when the selected classes have a status which enables the desired functionality to be applied. For example, the type checker and the pretty printer features are enabled only when the class has been accepted by the syntax checker.

The various actions are described in more detail in later sections as follows:

Syntax Check (

40

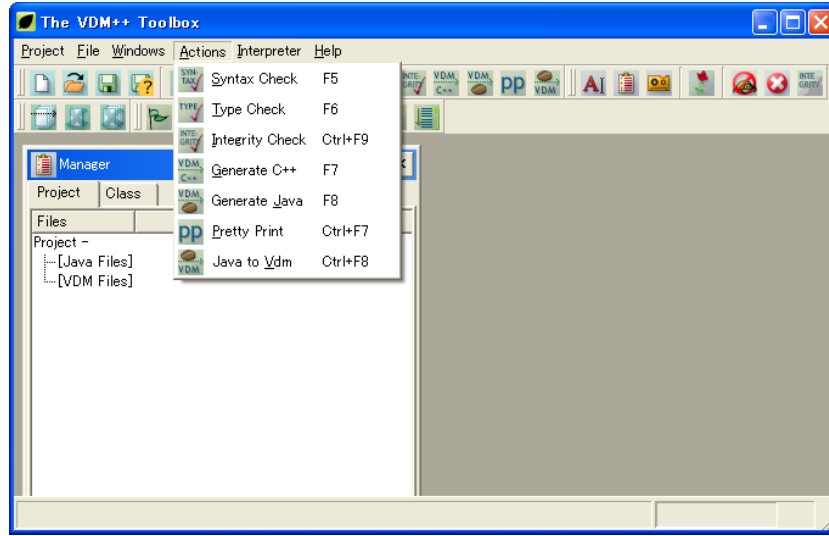




Figure 21: The Actions Menu and Toolbar

Type Check (): see Section 4.4

Generate Integrity Properties (): see Section 4.6


Generate C++ (): see Section 4.8

Generate Java (): see Section 4.9



Pretty Print (): see Section 4.7

Java to VDM (): see [1]

4.1.3 The log window, error list and source window

The Log Window displays messages from the Toolbox, including messages reporting on the success or failure of applying the actions described above. It opens automatically (if it is not already open) when a new message is displayed. Alternatively, it can be opened/closed by hand by pressing the  button on the Window Operations toolbar or by selecting the corresponding item from the Windows menu.

The Error List reports errors discovered by the Toolbox while performing actions. It has two panes as shown in Figure 22. The top pane shows a list of the places (file name, line number, column number) at which errors and warnings arose,

while the bottom displays a more detailed explanation of the currently selected error. The format of the various errors which can arise during syntax checking and type checking is described in Sections 4.3.2 and 4.4.2 respectively. Initially, the first error in the list is selected automatically. You can get to the next/previous reported error by pressing respectively the  or  button which appears to the left of the error list. Alternatively you can move to an arbitrary error by selecting the error notifier directly in the top pane of the Error List.

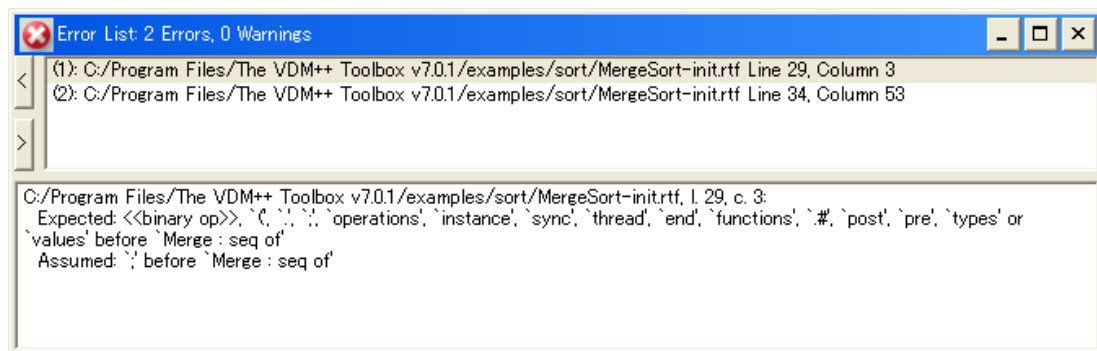







Figure 22: The Error List

The **Error List** opens automatically (if it is not already open) when a new error is discovered. Alternatively, it can be opened/closed by hand by pressing the  button on the **Window Operations** toolbar or by selecting the corresponding item from the **Windows** menu.

The **Source Window** also opens automatically (if it is not already open) when a new error is discovered. It displays the part of the source specification in which the currently selected error was discovered, the actual position of the error being marked by the window's cursor. The **Source Window** corresponding to the **Error List** illustrated in Figure 22 is shown in Figure 23.

Many source files can be present in the **Source Window** at the same time but only the contents of one of them is shown. The display can be changed to show a different source file by selecting the tab corresponding to that file at the top of the **Source Window**. New source files can be added to the display by hand by double-clicking the left mouse button on the file name (or on one of the classes contained in the file) in the **Manager**. Source files can be removed from the display by pressing either the  (Close file) button or the  (Close all files) button on the **File Operations** toolbar: the former () closes only the file which is currently visible, while the latter () closes all files.

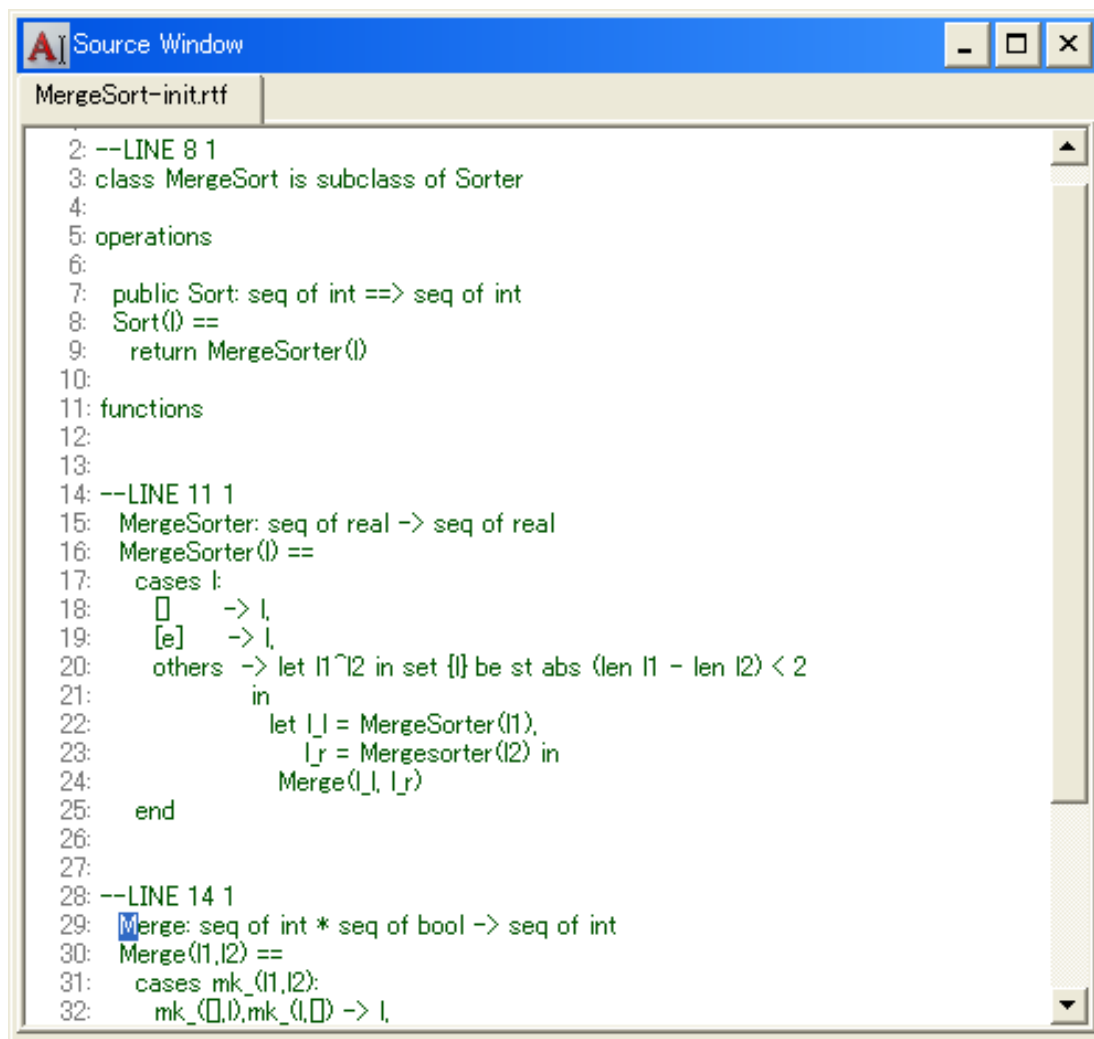




Figure 23: The Source Window

The Source Window can be opened/closed by hand by pressing the  button on the Window Operations toolbar or by selecting the corresponding item from the Windows menu.


4.1.4 Editing files

In order to allow you to fix errors reported by the Toolbox without leaving the Toolbox you can invoke your preferred editor (see Appendix C) directly on files

in the current project: simply select the appropriate file(s) in the **Manager** and press the **External Editor** button () on the (**Project Operations**) toolbar. Note that if more than one file is selected when you invoke the **External Editor** in this way you actually get one **External Editor** for each of the selected files.

The Toolbox automatically registers the changes that you make to the file(s) when you save them in the editor. However, the edited versions of the files are not re-processed automatically so whenever you edit a source file you must run the syntax checker on it again before the other tools in the Toolbox will be aware of the changes you have made.

4.1.5 Using the interpreter

The interpreter allows you to evaluate and debug expressions and statements. The **Interpreter Window**, which provides an interface to the interpreter, is opened by pressing the  (**Interpreter**) button on the (**Window Operations**) toolbar, and the **Interpreter** menu and toolbar offer a range of operations which can be applied in the interpreter. The interpreter is described in detail in Section 4.5.

4.1.6 On-line help

On-line help for the Toolbox and the interface in general can be accessed through the **Help** toolbar or the **Help** menu. Currently only the following very limited help is available:

About (): Displays the version number of the Toolbox.

aboutqt (): Displays information about and a reference to Qt, the multiplatform C++ GUI toolkit which the Toolbox interface uses.

4.2 The Overall Command Line Interface

The command line interface is started from a command prompt by typing¹³:

```
vppde specfile
```

¹³Either the executable **vppde** must be in the search path or the full path to it must be given as well.

When **vppde** is called from the command line without any options and with only one file argument (which must contain a VDM++ specification), the tool will enter the command mode and begin by syntax checking the argument file.

The user manipulates, executes and debugs a specification using a number of commands typed at the prompt produced by the Toolbox. The commands given below are supported by **vppde**. The abbreviations in parentheses are short forms for the commands.

A number of the commands cannot be called before the specification has been initialised (see the **init** command in Section 4.5). These commands are marked with a star (*).

A number of commands can be used to display the names of different constructs. These are, **classes**, **functions**, **operations**, **instvars**, **types** and **values**. Help for the Toolbox commands can be obtained using either **info** or **help**. Sequences of frequently used commands can be collected in script files and activated using the **script** command. General operating system calls can be made using the **system** command. The **dir** command can be used to add more directories to the search path used by the Toolbox. **pwd** gives the current working directory. Finally the **quit** and **cquit** commands can be used to leave the command line version of the Toolbox. These commands are described as follows:

***classes**

Displays the names of the defined classes and their status.

***functions class**

Displays the names of the functions defined in class **class**. Includes precondition, postcondition and invariant functions which are automatically created when the specification includes such constructs.

***operations class**

Displays the names of the operations defined in the given class.

***instvars class**

Displays the names of the instance variables of the given class.

***types class**

Displays the names of the types defined in the given class.

***values class**

Displays the names of the values defined in the given class.

help [command]

On-line help explaining all available commands in the same style as is used in this section. Without an argument it lists all the available commands. Otherwise the command **command** is described.

info [command]

Same as **help**.

script file

Reads and executes the script in **file**. A script is a sequence of VDM++ commands. These can be any of the commands described in this section and in other sections about the command line interface. When the script has been executed, the control is returned to the Toolbox.

system (sys) command

Executes a shell command.

dir [path ...]

Adds a directory to the list of active directories. These are the directories that will be searched automatically when trying to locate a specification file.

When calling this command with no arguments the list of active directories is printed to the screen. The directories will be searched in the displayed order.

pwd

Gives the current working directory i.e. the directory in which the current project file is placed (if a project file exists). In all cases this is the directory in which the **vdm.tc** file must be placed, and where files generated by the code generator and the Rose-VDM++ Link are written.

cquit

Quits the debugger without asking for confirmation. This is useful when using the debugger in a batch job.

quit (q)

Same as **cquit**.

4.2.1 Initialisation file

It is possible to put command line interface commands into an “initialisation file”. These commands will be executed automatically when the Toolbox is started from the command line.

The initialisation file must be called `.vppde` and must be located either in the directory from which the Toolbox is started or in the same directory as the specification file which is given as argument.


4.3 The Syntax Checker

The syntax checker checks whether your specification conforms to the syntax given in the language definition. The other tools in the system rely on the specification being syntax-correct, so your specification must have been syntax checked with no syntax errors before the other tools in the Toolbox can be applied. Note that when you change a source file you must syntax check it again before the other tools will be aware of the changes you have made.

The syntax checker can be accessed from either the graphical, command line or Emacs interface.

The syntax checker aims to report as many of the syntax errors in a specification as possible at the same time. Consequently, it uses an advanced recovery mechanism which allows it to detect and recover from a syntax error before passing on to report subsequent syntax errors in the specification. It does this either by ignoring some symbols in the specification or by assuming additional symbols. The error messages it gives include information about what was expected at an error point in the specification and what was ignored or assumed in order to allow the checker to carry on. Initially, it is easiest to understand the error messages by concentrating on what was assumed or ignored because this guess by the syntax checker is often close to the real error.

4.3.1 The graphical user interface

To start the syntax checker from the graphical user interface, select the files or classes (more than one, if you wish) you want to check or recheck in the **Project View** or the **VDM View** of the **Manager** as appropriate¹⁴, then press the  (Syntax Check) button on the (Actions) toolbar to invoke the syntax checker. The **Log Window** opens automatically (if it is not already open) and displays information about the checking process for each selected file or class in turn. If syntax errors are discovered, the **Error List** and the **Source Window** are also automatically invoked.

¹⁴If you select classes in the **VDM View** of the **Manager** the syntax checker is actually applied to the set of files which contain the selected classes – the Toolbox only knows which files have been edited. This of course means that if a particular file contains more than one class definition and you select only some of those classes then the other classes in the same file are implicitly included.

4.3.2 Format of syntax errors

When a syntax error in the specification is discovered the syntax checker displays information about the error in the **Error List** as follows:

1. It prints the symbols which were **expected** at the place of the syntax error.
2. It prints how it tried to recover from the syntax error, which could be by **inserting** one or more symbols, by **ignoring** one or more symbols, or by **replacing** some input symbols with other symbols at the point of the syntax error. The specification file is *not* changed by this operation, i.e. the change is only performed internally within the syntax checker to enable it to detect multiple syntax errors.

The symbols are displayed in a mixture of three formats:

- Display of text within single quotes, e.g. `'functions'`.
- Display of a meta-symbol, e.g. `<end of file>`, the designation of the end of the file.
- Display of a group of similar tokens as a single token, e.g. `<<type>>`, the syntactic unit `type` whose definition can be found in [4]. This is done to shorten the list of expected symbols.

4.3.3 The command line interface

The syntax of the command to invoke the syntax checker at the command line is:

```
vppde -p [-w] [-R testcoverage] specfile(s) ...
```

With the `-p` option, `vppde` syntax checks a number of files, each containing one or more classes. Syntax errors are reported to `stderr`.

The additional options that can be used with the syntax checker are:

- `-w` This option causes the Toolbox to write the VDM++ parts of RTF files in ASCII files. The names of these ASCII files will be the RTF file names with the extra extension `.txt`, e.g. `sort.rtf` will yield `sort.rtf.txt`.

This option is typically used in a test environment to reduce the time used to parse specification files. If the documentation parts of RTF files are very large this can slow down the parsing since the entire file must be parsed. For example, figures tend to make the documentation part of a file very large.

- R Causes the Toolbox to produce a test coverage file **testcoverage** which is used to keep track of how often different constructs have been exercised during testing of a VDM++ specification. In the current version this test coverage file must be called **vdm.tc** for the pretty printer to work. See Section 4.10 for an example.

4.3.4 The Emacs interface

In the Emacs interface all commands are given at the command prompt. Syntax checking is made by the **read** command and traversing the syntax errors is done using the **first**, **last**, **next** and **previous** commands. The location of the errors is shown in the specification window. In more detail, the commands are:

read (r) file(s)

Syntax checks specifications from **file(s)** The file(s) must contain definitions of classes including operations, functions, values, types, and instance variables.

The contents of each file is treated as a whole. This means that if a syntax error occurs then none of the VDM++ constructs in the file are included. This is also the case if the file contains more than one class (i.e. none of the classes are included). If a file is syntax checked successfully and redefines a class which is already defined in a syntax checked file then a warning is given.

first (f)

This command displays the position of the first recorded error or warning message from the syntax checker, type checker, code generator or pretty printer.

last

This command displays the position of the last recorded error or warning message from the syntax checker, type checker, code generator or pretty printer.

next (n)

This command displays the position of the next recorded message in the source file window. It is used to display error or warning messages from the syntax checker, type checker, code generator and pretty printer.

previous (pr)

This command displays the position of the previous recorded message. It is also used to display error or warning messages from the syntax checker, type checker, code generator and pretty printer.

4.4 The Type Checker

The type checker assesses whether expressions are of the types expected for their positions in a specification. However, type correctness is not always as clear-cut as it seems. For example, if a function takes an `int` as argument but is applied to an expression of type `real`, then, since `int` is a subtype of `real`, the application might be correct provided the function is called at run-time with actual parameters which happen to be integer reals. On the other hand, the application might also be incorrect since `real` contains elements that are not part of `int`. We say that such an application is *possibly* well-formed but not *definitely* well-formed.

In fact the type checker can perform type checking at either of these two different levels: possible and definite well-formedness. In short the difference between them is that specifications which are possibly well-formed can be type correct but are not guaranteed to be so, whereas specifications that are definitely well-formed are guaranteed to be type correct. Thus, the function application discussed in the previous paragraph would pass a possible well-formedness (“pos”) type check but fail a definite well-formedness (“def”) type check: the “def” check would identify it as a possible source of a run-time error.


The definite well-formedness check will identify all places where run-time errors could potentially occur. These include applications of functions which have a precondition (the precondition must be satisfied before an application of that function is made) and applications of partial operators which are built directly into VDM (e.g. the arithmetic division operator which gives a run-time error if its second argument is zero), as well as possible inconsistencies resulting from the use of a subtype in a definition, either through an invariant or through the use of one part of a union type.

In general a “def” type check will yield more error messages than a “pos” type check. Therefore we recommend that you always run a “pos” check on your specification first in order to deal with all the points where the specification is not even possibly type correct, then run the “def” check in order to identify possible causes of run-time errors. In many cases, you will be able to eliminate these from consideration, for example because an expression is protected by being in one limb of an “if ... then ... else ...” expression where the condition prevents the run-time error condition from arising. In other cases, the “def” check may identify conditions for which you do want to introduce protection by modifying the specification.


The type checker can be accessed either from the GUI, from the command line

version of the Toolbox, or from the Emacs interface.

4.4.1 The graphical user interface

In order to invoke the type checker from the graphical user interface select the files or classes (more than one, if you wish) to be checked or rechecked in the **Project View** or the **VDM View** of the **Manager** as appropriate, then press the  (**Type Check**) button on the (**Actions**) toolbar. The **Log Window** opens automatically (if it is not already open) and displays information about the checking process for each selected file or class in turn. If type errors are discovered, the **Error List** and the **Source Window** are also automatically invoked. Note that since the Toolbox knows the dependencies between all classes, all the super classes of the selected classes will also be type checked.

Setting options

The choice between checking for possible or definite type well-formedness is made in the **Type checker** pane of the **Project Options** window, which is displayed by pressing the  (**Project Options**) button on the (**Project Operations**) toolbar. This is shown in Figure 24. Either “pos” type checking or “def” type checking will always be enabled. The default is possible well-formedness checking.

Two further options are also offered:

Extended type check: If enabled, a number of additional warnings such as “Result of ‘conc’ can be an empty sequence” will be included when type checking.
default: disabled.

Warning/error message separation: If enabled, separates error messages and warnings from the type checker when they are displayed in the **Error List**: error messages are displayed before warnings.
default: enabled.

4.4.2 Format of type errors and warnings

All warnings provided by the type checker are textual descriptions explaining what the potential problem is. Some errors such as unknown identifiers are also simply textual. However, the majority of type errors are structured into three

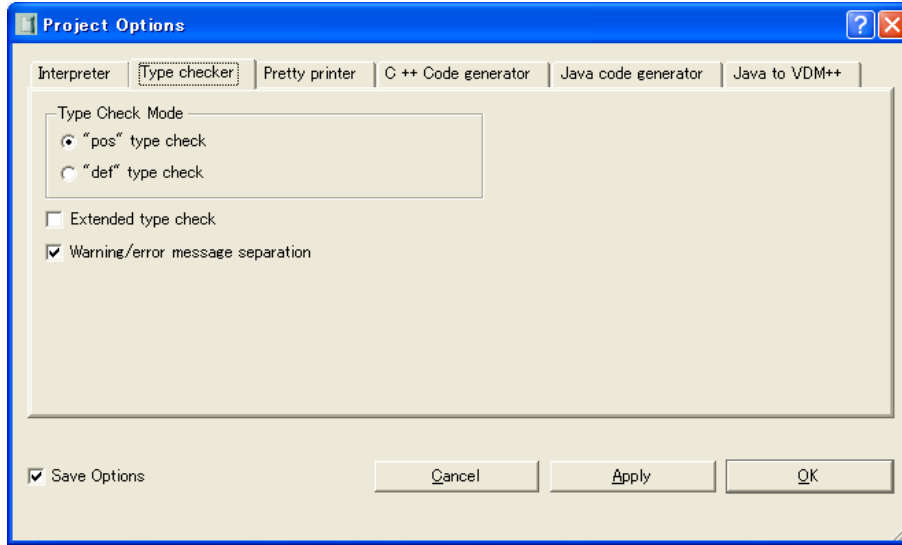


Figure 24: Setting Type Checker Options

lines, in which the first line gives a textual explanation about what the problem is, the second line gives the actual type inferred by the type checker (identified by the keyword `act:`), and the third line gives the type expected by the type checker (identified by the keyword `exp:`). The syntax for these type descriptions is almost identical to the normal VDM++ type syntax with the following exceptions:

- `seq of A` is represented as `seq1 of A | []`, where `[]` is the type for an empty sequence.
- `map A to B` is represented as `map A to B | {|->}`, where `{|->}` is the type for an empty map.
- `set of A` is represented as `set of A | {}`, where `{}` is the type for an empty set.
- `[A]` is represented as `A | nil`.
- `#` stands for any type. The type checker typically infers this type if it cannot infer anything better in an error situation.

Examples of type errors can be found in Section 3.7.

Understanding errors from “def” type check

Recall that the “def” check produces an error report wherever it is not possible to guarantee that an expression will always be of the correct type. In order to understand some of the warning and error messages generated during a check for definite well-formedness, it can often be helpful to insert the word ‘DEFINITELY’ in the error message implicitly. Thus, for example, if the message

Error : Pattern in Let-Be-expression cannot match

is returned in a check for definite well-formedness you should read it as

Error : Pattern in Let-Be-expression cannot DEFINITELY match

i.e. that there could be values for which the pattern may not match. When the type checker reports an error, it will often display which type it inferred and which type it expected at a given point. This can be valuable when trying to find out what is wrong.

4.4.3 The command line interface

```
vppde -t [-df] specfile(s) ...
```

With the `-t` option `vppde` type checks the `specfile(s)`. First, the specification is parsed. Then, if no syntax errors are detected, the specification is type checked (the default is to check for possible well-formedness). Type errors are reported to `stderr`.

The additional options which can be used with the type checker are:

- d Causes the type checker to check for definite well-formedness. The difference between possible and definite well-formedness is described in the language reference manual ([4]). In short the check for definite well-formedness returns the type-related proof obligations.
- f Causes the type checker to perform an extended type check. This will give some extra warning and error messages for both possible and definite well-formedness checks such as “Result of ‘conc’ can be an empty sequence”.

4.4.4 The Emacs interface

In the Emacs interface all commands are given at the command prompt. Type checking is performed by the **typecheck** command and traversing the warnings and type errors is done using the **first**, **last**, **next** and **previous** commands, as for syntax errors. The location of the errors is shown in the specification window. The extended type check option for the type checker can be enabled using the **set** command and disabled using the **unset** command. In more detail, the available commands are as follows:

typecheck (tc) class option

This command makes a static type check of the given class. (“*” is used to do class all current directories in type check.) The **option** can be either **pos** or **def**, indicating whether the specification should be checked for possible or definite well-formedness.

If a type error occurs it is reported, with position information, in the specification window.

first (f)

This command displays the position of the first recorded error or warning message from the syntax checker, type checker, code generator or pretty printer.

last

This command displays the position of the last recorded error or warning message from the syntax checker, type checker, code generator or pretty printer.

next (n)

This command moves the current position to the next recorded error or warning message in the source file window. It is used to display error or warning messages from the syntax checker, type checker, code generator or pretty printer.

previous (pr)

This command moves the current position to the previous recorded error or warning message in the source file window. It is also used to display error or warning messages from the syntax checker, type checker, code generator or pretty printer.

set full

The command **set** enables setting of the internal options of the Toolbox. If the command is called without parameters it displays the current settings.

full enables extended type checks. This option has effect for both possible and definite well-formedness checks. By default this option is disabled.

unset full


Disables the extended type checks.

4.5 The Interpreter and Debugger

The interpreter and debugger enable execution of VDM++ specifications. It is not necessary to have type checked any classes before the interpreter can be used (but naturally more run-time errors are likely to occur when a specification is not type correct). The interpreter/debugger can be accessed from either the GUI, the command line interface or the Emacs interface.

The only VDM++ constructs that cannot be executed are implicitly defined functions and operations, specification statements, type bindings, and expressions conforming to the restrictions that our modelling of the VDM++ three-valued logic impose. Support for the concurrency and real-time parts of VDM++ is also not yet available within the interpreter. These constructs are described further in [4].

4.5.1 The graphical user interface

The **Interpreter Window** can be opened/closed by pressing the  (Interpreter) button on the (Window Operations) toolbar or by selecting the corresponding item from the **Windows** menu.










The top two panes of the tool are respectively the **Response** and **Dialog** panes: you can give commands directly to the **Interpreter** in the **Dialog** pane and you receive output from the interpreter in the **Response** pane. To evaluate a VDM++ expression, you type it directly on the command line in the **Dialog** pane.

The two panes at the bottom of the tool are the **Trace** and the **Breakpoints** panes. The first of these shows the function/operation call stack which logs the various function/operation calls made as well as the actual parameters to each of those calls. The parameters are generally elided by default and just appear in the form ‘...’. They can be revealed by clicking the left mouse button on the ‘...’. Clicking the left mouse button on the revealed parameters will replace them with ‘...’ again.


The **Breakpoints** pane shows a list of the locations of all the current breakpoints together with their status, which may be **enabled** (indicated by a ☒ to the left of the function/operation name) or **disabled** (indicated by a ☐ to the left of the function/operation name). The buttons at the top of the pane respectively enable, disable, or delete the breakpoints currently selected in this list.

The **Interpreter** menu and toolbar offer a range of operations which can be applied

in the interpreter:






- Init** (): Initialise the specification. This means that all the global values and instance variables are initialised. Initialisation of the interpreter must be made first to enable use of the definitions which have been syntax checked.
- Step** (): Execute the next statement, without stepping into function and operation calls, and then break. This button is not useful with functions because it evaluates the entire body expression.
- Step In** (): Execute the next expression or statement, including stepping into function and operation calls, and then break.
- Single Step** (): Execute the next subexpression or substatement, without stepping into function and operation calls, and then break.
- Continue** (): Use this to continue execution after a breakpoint until the next breakpoint or the end of the expression/statement evaluation is reached.
- Finish** (): Finish the evaluation of the current function or operation and return to the caller. The command is traditionally used together with **Step In**.
- Up** (): This command can only be called after the specification has been initialised and the debugger has stopped at a breakpoint. It has the effect that the current context is shifted one level up compared to the place currently shown in the display window. Thus, the context is changed to the place in the current function trace where the current function/operation was called.
- Down** (): This command can only be called after the specification has been initialised and the debugger has stopped at a breakpoint. It has the effect that the current context is shifted one level down compared to the place currently shown in the display window. Thus, the context is changed to the place in the current function trace where the current function/operation called its sub-function/operation.
- Stop** (): Stop the evaluation of an expression. Access to local and global variables depends on whether the button has been pressed within a **print** or a **debug** command (see the description of these commands below for a description of this). The command is traditionally used to break a possible infinite loop in one's specification.

Commands available in the dialog pane

In addition to the operations described above, commands to the interpreter can be input directly by typing them in the **Dialog** pane. These are described below. However, a number of these commands can only be executed after the interpreter has been initialised (by pressing the  (Init) button). These are marked with a star (*).

An expression can be evaluated using either the **print** or the **debug** command. The only difference between the two commands is that **debug** causes the interpreter to stop at breakpoints whereas **print** ignores breakpoints.

Breakpoints can be set using the **break** command or by double-clicking on the desired position in the **Display** window¹⁵.

When a break point is reached it is possible to continue the execution using either the **Step** () , **Single Step** () , **Step In** () , **Continue** () or **Finish** () buttons to proceed with the execution. Breakpoints can be deleted using the **delete** command.

Objects can be created using the **create** command and destroyed using the **destroy** command. The **objects** command gives a list of the names of all the current objects.

There are also three commands relating to threads: the identifier of the current thread (during an execution) can be obtained using the **curthread** command; a list of all the threads currently executing can be obtained using the **threads** command; and a different thread can be selected using the **selthread** command.

In addition to these commands, which are explained in more detail below, Section 4.2 also contains a number of commands which are useful in the **Dialog** window.

The up-arrow and down-arrow keys can be used to scroll through previous commands. Pressing enter in this history list will execute the corresponding command. If some characters have been written before beginning to scroll through the history list only those previous commands that start with these exact characters are shown.

Pressing enter without typing a new command executes the previous command.

¹⁵When the RTF format is used double-clicking does not work. Instead one must press Ctrl-Alt-Spacebar on the line where one wishes to break inside Microsoft Word.

***break (b) [name]**

Sets a breakpoint at the function or operation with the given name. The name must consist of the function/operation name qualified with the name of the class in which it is defined (i.e. in the form **ClassName** '**OperationName**').

When this command is evaluated a number is allocated for the new breakpoint and this is shown in the **Response** pane. The name and number of the new breakpoint are also added to the list of breakpoints in the **Breakpoints** pane.

If called with no argument, it displays a list of all the currently defined breakpoints.

***break (b) name number [number]**

This sets a breakpoint on the line with the given number in the file with the given name. If a second number is given, this is interpreted as the column at which the breakpoint should be set.

Note that if the source file is not an RTF file you can also set breakpoints by double-clicking the left or middle mouse button on the desired position in the file in the **Source Window**. If you are using an RTF source file you can similarly set a breakpoint by positioning the cursor at the appropriate position in the file in Microsoft Word, then pressing **Control-Alt-spacebar**¹⁶.

***create (cr) name := stmt**

This command creates an object reference of name **name** initially assigned to **stmt**. **stmt** must be either a call statement referring to an object or a new statement. (See [4] for an explanation of the different kinds of statements.) Afterwards the object **name** will be in the scope of the debugger.

curthread

Prints the identifier of the thread currently being executed.

debug (d) expr

Evaluates and prints the value of the VDM++ expression **expr**. The execution will be stopped at all enabled breakpoints with the current position of the execution being displayed in the **Source Window** and the call stack being shown in the **Trace** pane. If a run-time error occurs, the execution is stopped in the context where the error occurred with the position of the error being displayed in the **Source Window** window and the call stack being shown in the **Trace** pane.

¹⁶This works with versions of the VDM template, VDM.dot, distributed with Toolbox version v9.0.6 and onwards.

When evaluating an expression in the interpreter you can use the symbol **\$\$** to refer to the result of the last evaluation. See the description of the **print** command for more information.

If the **Stop** button is pressed during a debug command the evaluation of the command is stopped at the expression or statement being evaluated when the button is pressed. All the variables within scope of that expression or statement can be accessed afterwards.

***delete number, ...**

Deletes the breakpoint(s) with the given number(s). The breakpoints are also removed from the **Breakpoints** pane.

***destroy name**

Destroys the object with the given name.

***disable number, ...**

Disables the breakpoint(s) with the given number(s).

***enable number, ...**

Enables the breakpoint(s) with the given number(s).

init (i)

Initialises the interpreter with all definitions from the specification. This includes initialising the instance variables and all values. If a value is multiply defined this will be reported during this initialisation. The initialisation command will initialise all files read into the Toolbox in the same session. Therefore it is not necessary to initialise each file separately after it has been read.

***objects**

Displays the objects created within the debugger.

***popd**

This command is used when nested debugging is taking place i.e. when an expression is debugged while already at a breakpoint in another evaluation. The effect of a **popd** command is to restore the environment to that which existed when the last **debug** command was invoked.

print (p) expr, ...

Evaluates and prints the value of the VDM++ expression(s) **expr** with all breakpoints disabled. If a run-time error occurs the execution stops and the position of the error is displayed in the **Source Window**.

In addition to the normal VDM++ values the **print** command can also return the values **FUNCTION_VAL** and **OPERATION_VAL**. This happens if the

result of the evaluation is a function or an operation (for example if a function is evaluated just by giving the function name without supplying any parameters enclosed in parentheses).

When evaluating an expression in the interpreter you can use the symbol `$$` to refer to the result of the last evaluation. This symbol is treated as an expression and can therefore be embedded in other VDM++ expressions as shown in the following examples:

```
vdm> p 10
10
vdm> p $$+$$, 2*$$
20
40
vdm>
```

If the **Stop** button is pressed during a print command the evaluation of the command is stopped. No variables can be accessed afterwards.

priorityfile (pf) [filename ...]

If called with a valid filename, this reads the priority information from this file and uses it when scheduling threads if priority-based scheduling is enabled.

If called with no argument it lists the current priority file being used by the interpreter.

See Appendix [G](#) for details of the required format for priority files.

***push name**

The class **name** is pushed onto the modules stack and becomes the active class after initialisation.

***pop**

The current class is popped off the stack. If there is no active class a warning is issued and nothing happens.

selthread id

Sets the currently executing thread to be that with identifier **id**.

threads

Displays a list of the threads currently being executed in the following format:

< thread id > < object ref > < status >

where *thread id* is the unique identifier of the thread, *object ref* is the identifier of the object within which the thread is defined (**none** if this is the thread of control initiated by the interpreter), and *status* is one of the following:

Status	Meaning
Blocked	the thread is waiting for a permission predicate to become true.
Stopped	the thread has stopped at a breakpoint.
Running	the thread is currently being executed by the interpreter.
Sleeping	the thread can be scheduled, but has actually not started running yet.
MaxReached	the maximum number of instructions per time slice has been reached by this thread.

tcov

The test coverage command **tcov** makes it possible to control the collection of test coverage information. It is used in combination with various keywords as follows:

tcov read filename

Reads the test coverage information saved in the given file.

Note that if you syntax check a file after reading in a test coverage file the coverage information for that file will be reset and the test coverage information will be lost unless you write the test coverage information before the file is syntax checked. Also be aware that the pretty printing function always uses the test coverage file that is specified in the specification file.

tcov write filename

Writes the existing test coverage information to the given file.

tcov reset

Resets all test coverage information to zero.

Setting options

The interpreter has a number of options which can be set in the **Interpreter** pane of the Project Options window (see Figure 25). These options are:

Dynamic type check: If this check is enabled the type of expressions will be checked according to the definition given in the VDM++ specification whenever a

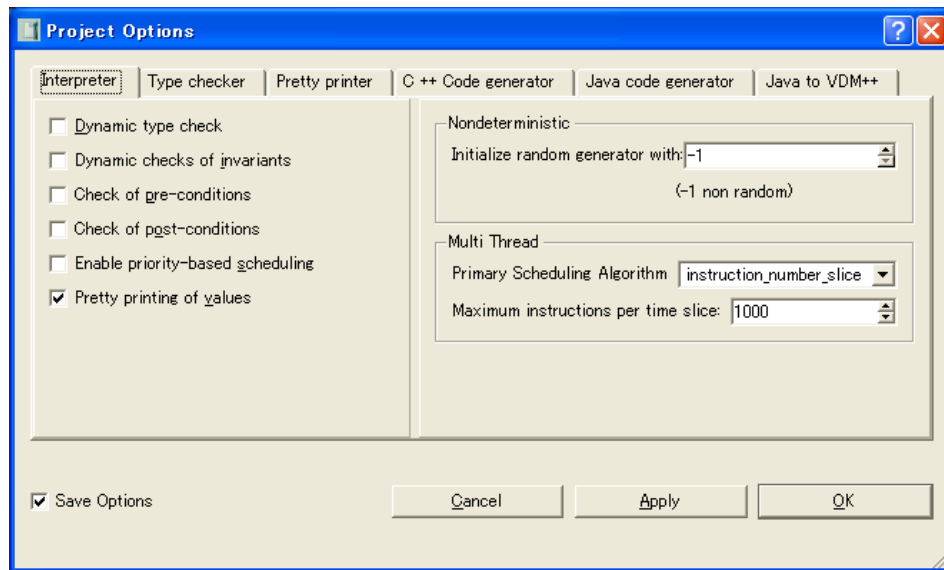


Figure 25: Setting Interpreter Options

type has been fixed.

default: disabled.

Dynamic checks of invariants: If this check is enabled, expressions will be checked against the invariant on their type whenever such an invariant exists.
default: disabled.

Check of pre-conditions: If this check is enabled the precondition of every function which is evaluated will be checked before the function is called.
default: disabled.

Check of post-conditions: If this check is enabled the postconditions of every function and operation which is evaluated will be checked after the function or operation has been evaluated.
default: disabled.

Pretty printing of values: Causes the pretty printer to use a nice, easy-to-read style for printing values in which line breaks and a homogeneous indentation are inserted.
default: enabled.

Initialise random generator with: Initialises a random number generator with the given integer. This causes a random order of evaluation of substatements in non-deterministic statement constructs. The integer must be larger than

or equal to zero. A negative number disables random evaluation of non-deterministic statements.

default value: -1.

Enable priority-based scheduling: enables use of priority-based scheduling instead of round-robin scheduling.

default: disabled

Maximum instructions per time slice: sets the number of instructions per time slice to be the given integer (used for instruction number slice scheduling).

default value: 1000.

Primary Scheduling Algorithm: Sets the primary scheduling algorithm to be either pure cooperative scheduling or instruction number slicing.

default: Instruction number slicing.

4.5.2 Standard libraries

Currently there are three standard libraries: one for VDM utilities, one for maths and one for input/output functionality.

The VDMUtil Library

The interpreter provides a VDMUtil standard library. The functions and values available and their concrete syntax are described in [5]. To use this library the file `VDMUtil.vpp` must be part of the project. The file is located in the `vpphome/stdlib` directory.

The `VDMUtil.vpp` file contains a number of functions that are all defined as is not yet specified. In a general VDM++ specification such functions cannot be executed by the interpreter, but for these particular functions definitions exist within the toolbox. Thus, if you include the `VDMUtil.vpp` file in your project these utility functions will be available with your specification.

The Maths Library

The interpreter provides a maths standard library. The functions and values available and their concrete syntax are described in [5]. To use this library the file

`math.vpp` must be part of the project. The file is located in the `vpphome/stdlib` directory.

The `math.vpp` file contains a number of functions that are all defined as **is not yet specified**. In a general VDM++ specification such functions cannot be executed by the interpreter, but for these particular functions definitions exist within the toolbox. Thus, if you include the `math.vpp` file in you project these maths functions will be available with your specification.

The IO Library

The interpreter provides an IO (input/output) standard library. The functions and values available and their concrete syntax are described in [5]. To use this library the file `io.vpp` must be part of the project. The file is located in the `vpphome/stdlib` directory.

The `io.vpp` file contains a number of functions that are all defined as **is not yet specified**. In a general VDM++ specification such functions cannot be executed by the interpreter, but for these particular functions definitions exist within the toolbox. Thus, if you include the `io.vpp` file in you project these IO functions will be available with your specification.

4.5.3 The command line interface

The interpreter/debugger is invoked by the following command:

```
vppde -i [-O res-file] [-R testcoverage] [-D [-I]] [-P] [-Q]
        [-Z priority-file] [-M num] argfile specfiles
```

With the `-i` option `vppde` evaluates a VDM++ expression (or a sequence of VDM++ expressions separated by commas) in the file `argfile` in the context of the specification in the `specfile(s)`. The result of the evaluation is reported to `stdout`. When a sequence of expressions is used, it is possible to refer to the result of the previous expression by writing `$$`.

If a run-time error is encountered, the interpretation is terminated and an error message is displayed. The error message contains position information for the construct that caused the error and a message describing the type of error.

The additional options that can be used with the interpreter are:

- D Enables dynamic type checking.
- I Enables invariant checking. This option only has effect if the -D option is enabled as well.
- P Enables precondition checking for all functions which are evaluated.
- Q Enables postcondition checking for all functions and operations which are evaluated.
- R The result of the interpretation will be the same as if the argument file was evaluated with the specification files used for generating the `testcoverage` file. The difference is that the interpreter will update the `testcoverage` file with run-time information and save it to the hard disk after the evaluation. See Section 4.10 for an example.
- O `res-file` Prints the result of evaluating the `argfile` to the `res-file`. If `res-file` already exists it will be overwritten. This option is typically used in test scripts in which the result is automatically compared with expected results.
- Z `priority-file` Evaluate using priority based scheduling. Only has effect if used with a multi-threaded model.
- M `num` Use `num` as the number of instructions per slice.
- S `algorithm` Use the specified scheduling algorithm. This may be one of:
 - `pure_cooperative` Pure cooperative scheduling;
 - `instruction_number_slice` Instruction number sliced scheduling.

4.5.4 The Emacs interface

In the Emacs interface all commands are given at the command prompt. Initialisation of the interpreter must be made first to enable use of the definitions which have been syntax checked. This is done using the **init** command. A number of commands cannot be called before the specification has been initialised (see the **init** command below). These commands are marked with a star (*).

An expression can be evaluated using either the **print** or the **debug** command. The only difference between the two commands is that **debug** will force the interpreter to stop at breakpoints whereas **print** ignores breakpoints. Breakpoints

can be set using the **break** command. When a breakpoint is reached it is possible to continue the execution using either the **step**, **singlestep**, **stepin**, **cont** or **finish** commands to proceed with the execution. Breakpoints can be deleted using the **delete** command.

The **backtrace** command can be used to inspect the call stack. Options to the interpreter can be set using the **set** command and they can be reset using the **unset** command.

New objects can be created using the **create** command and then can be destroyed again using the **destroy** command. The names of the current objects can be seen using the **objects** command.

***backtrace (bt)**

Displays the function/operation call stack.

***break (b) [name]**

Sets a breakpoint at the function or operation with the given name.

The name must consist of the function/operation name qualified with the name of the class it is defined in.

A number is allocated for the breakpoint and this is displayed as the result of the command.

If **break** is called without parameters it displays all the current breakpoints.

***break (b) name number [number]**

This sets a breakpoint in the file with the given name at the line with the given number. If a second number is given, this is interpreted as the column at which the breakpoint should be set.

***create (cr) name := stmt**

This command creates an object reference of name **name** initially assigned to **stmt**. **stmt** must be either a call statement referring to an object or a new statement. (See [4] for an explanation of the different kinds of statements.) Afterwards the object **name** will be in the scope of the debugger.

***cont (c)**

Continues execution after a breakpoint until another breakpoint or the end of the evaluation is reached.

curthread

Prints the identifier of the thread currently being executed.

debug (d) expr

Evaluates and prints the value of the VDM++ expression/statement **expr**. The execution stops at enabled breakpoints and the breakpoint position is displayed. If a run-time error occurs, the execution stops in the context where the error occurred and the position of the error is displayed in the specification window.

In order to use the last evaluated result in a following expression, **\$\$** can be used. See the description of the **print** command for more information.

***delete name ...**

Deletes the breakpoint set at the function(s) or operation(s) with the given name(s). Function and operation names must be qualified with the name of the class they are defined in.

***destroy name**

Destroys the object referred to by **name**.

***disable number**

Disables the breakpoint with the given number.

***enable number**

Enables the previously disabled breakpoint with the given number.

***finish**

Finishes the evaluation of the current function or operation and returns to the caller. The command is traditionally used together with **stepin**.

init (i)

This command initialises the interpreter with all definitions from the specification. This includes initialising the instance variables and all values. If a value is multiply defined this will be reported during this initialisation. The initialisation command will initialise all files read into the Toolbox in the same session. Therefore it is not necessary to initialise each file separately after it has been read in using the **read** command.

***objects**

Displays the objects created within the debugger.

***pop**

The current class is popped off the stack. If there is no active class a warning is issued and nothing happens.

***popd**

This command is used when nested debugging is taking place i.e. when an

expression is debugged while already at a breakpoint in another evaluation. The effect of a **popd** command is to restore the environment to that which existed when the last **debug** command was invoked.

print (p) expr,...

Evaluates and prints the value of the VDM++ expression(s) **expr** with all breakpoints disabled. If a run-time error occurs the execution will be stopped and the position of the error is displayed.

In addition to the normal VDM++ values the **print** command can also return the values **FUNCTION_VAL** and **OPERATION_VAL**. This happens if the result of the evaluation is a function or an operation (for example if a function is evaluated just by giving the function name without supplying any parameters enclosed in parenthesis).

In order to use the last evaluated result in a following expression, **\$\$** can be used. **\$\$** is treated as an expression and can therefore be embedded in other VDM++ expression(s) as shown in the following example:

```
vdm> p 10
10
vdm> p $$+$$, 2*$$
20
40
vdm>
```

priorityfile (pf) [filename ...]

If called with a valid filename, this reads the priority information from this file and uses it when scheduling threads, if priority-based scheduling is enabled.

If called with no argument, this lists the current priority file being used by the interpreter.

See Appendix [G](#) for further details of required format for priority files.

***push name**

The class **name** is pushed onto the modules stack and becomes the active class after initialisation.

remove number

Removes the breakpoint with the given number.

selthread id

Sets the currently executing thread to be that with identifier **id**.

set option

Enables setting of the internal options of the interpreter. If the command is called without parameters it displays the current settings. **option** can be one of the following:

dtc enables dynamic type checking.

inv enables dynamic checks of invariants. In order for **inv** to have any effect, **dtc** must be enabled as well.

pre enables check of preconditions.

post enables check of postconditions.

ppr enables pretty print format. All values will be displayed with line-breaks and indentation according to the structure of the value.

seed integer initialises a random number generator with the given integer. This causes a random order of evaluation of sub-statements in nondeterministic statement constructs. The integer must be ≥ 0 . A negative number will disable random evaluation of nondeterministic statements.

primaryalgorithm string sets the primary scheduling algorithm used by the interpreter to be the given string. This string may be:

pure_cooperative (pc) - use pure cooperative scheduling;

instruction_number_slice (in) - use instruction number slicing scheduling

Here the names in parentheses are abbreviations which may be used. See Section 4.5.5 for more details of the different scheduling algorithms. Default is **instruction_number_slice**

maxinstr integer use the given integer as the maximum number of instructions per slice. See Section 4.5.5 for details of how this value is used. Default is 1000.

priority enables priority-based scheduling. Section 4.5.5 for details of how this is used.

All options are false by default, except **ppr**.

***singlestep (g)**

Executes the next expression, sub-expression or statement and breaks.

***step (s)**

Executes the next statement and breaks. This command will not step into function and operation calls. This command is not useful with functions because it evaluates the entire expression.

***stepin (si)**

Executes the next expression or statement and then breaks. This command will also step into function and operation calls.

threads

Displays a list of the threads currently being executed in the following format:

< thread id > < object ref > < status >

where *thread id* is the unique identifier of the thread, *object ref* is the identifier of the object within which the thread is defined (**none** if this is the thread of control initiated by the interpreter), and *status* is one of the following

Status	Meaning
Blocked	the thread is waiting for a permission predicate to become true.
Stopped	the thread has stopped at a breakpoint.
Running	the thread is currently being executed by the interpreter.
MaxReached	the maximum number of instructions per time slice has been reached by this thread.

tcov

The test coverage command **tcov** makes it possible to control collection of test coverage information:

tcov read filename

Reads the test coverage information saved in the given file.

Note, that if you syntax check a file after reading in a test coverage file, the coverage information for that file will be reset and the test coverage information will be lost unless you write the test coverage information before the file is syntax checked. Also be aware that the pretty printing function always uses the test coverage file that is specified in the specification file.

tcov write filename

Writes the existing test coverage information to the given file.

tcov reset

Resets all test coverage information to zero.

unset option, ...

Disables one or more of the internal options of the Toolbox. See the **set** command for a description of the possible options.

4.5.5 Scheduling of threads

The following different primary scheduling algorithms are available:

Pure Cooperative Under this algorithm a thread will be executed until:

- It completes successfully;
- It reaches an operation call for which the corresponding permission predicate is false;
- A breakpoint is met or the interpreter is interrupted.

Instruction number slicing Under this algorithm a thread will be executed until:

- It completes successfully;
- It reaches an operation call for which the corresponding permission predicate is false;
- The number of (internal) instructions it has executed since being scheduled exceeds the `maxinstr` constant.
- A breakpoint is met or the interpreter is interrupted.

Selection of which thread to schedule next (secondary scheduling algorithm) follows a simple round-robin strategy, which may optionally be priority-based (set using the `Enable priority-based scheduling` option; see Section 4.5.1).


Note that if priority-based scheduling is used, the main thread (i.e. the thread initiated by the user) *always* has highest priority, above and beyond any priorities specified in the priority file.

4.6 The Integrity Examiner

The integrity examiner analyses the specification looking for places where run-time errors could potentially occur and generates a series of integrity properties which, if true, are sufficient to ensure that no run-time errors should occur. In all thirty different types of integrity properties are checked by the examiner.

These integrity properties are presented as VDM++ predicates that involve quantification over all possible values of the appropriate variables¹⁷, which means that if it can be demonstrated that an integrity property is true there will not be run-time errors associated with that integrity check whatever the values of the variables involved. Of course if an integrity property can instead be shown to be false this would point to there being a potential problem with the corresponding part of the specification.

The integrity examiner can only be accessed from the graphical interface.

To run the integrity examiner, select the files or classes (more than one, if you wish) you want to run it on in the **Project View** or the **VDM View** of the **Manager**, then press the  (**Generate Integrity Properties**) button on the (**Actions**) toolbar. The **Log Window** opens automatically (if it is not already open) and displays information about the examination process for each selected file or class in turn, and the **Integrity Properties Window** opens and displays the integrity properties generated. The **Integrity Properties Window** is shown in Figure 26.

The top pane of the **Integrity Properties Window** shows a list of the integrity properties together with information about their status (the **Checked** column), their position in the specification (the **Module**, **Member** and **Location** columns) and their type (the **Type** column). The numbers in the **Index** column simply serve to distinguish different integrity properties which have the same position. Clicking on a list heading orders the properties based on that particular attribute.

Selecting a particular integrity property in the top pane of the window causes the corresponding VDM++ predicate to be displayed in the bottom pane of the window. At the same time the cursor in the **Source Window** indicates the exact point in the specification to which the selected integrity property relates. Each integrity property can thus be inspected in order to try to determine whether or not it is true. For a more detailed explanation of this, see the example in Section 3.8.4.

¹⁷In some cases the full context is not shown explicitly and the scope of some variables has to be determined by inspection of the specification.

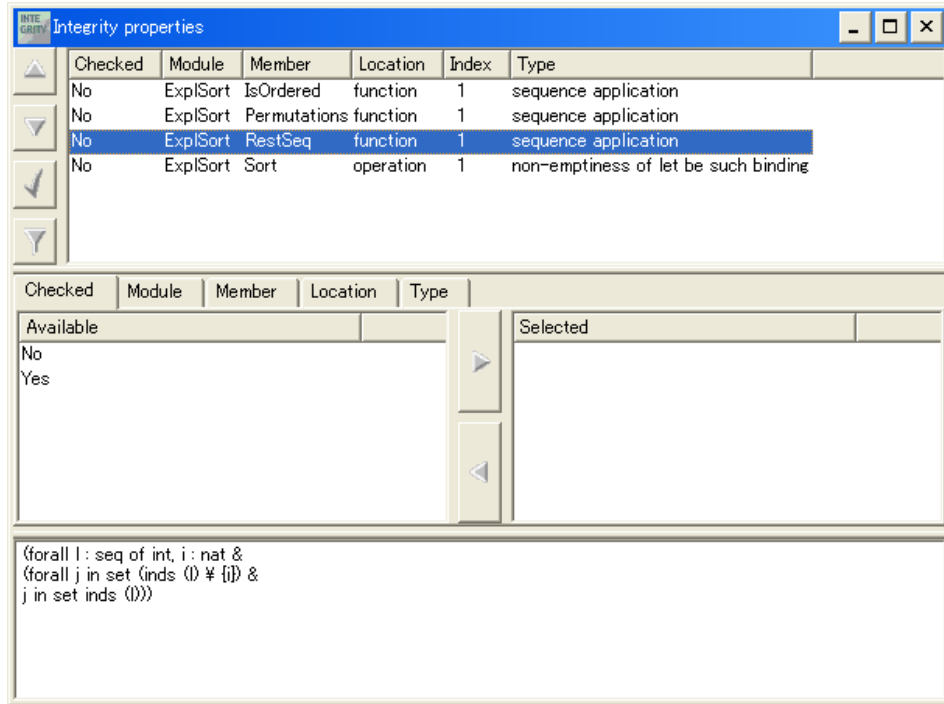










Figure 26: The Integrity Properties Window

The buttons  and  on the left-hand side of the top pane move the selection to the previous or the next integrity property respectively, while the  (**Toggle Status**) button toggles the status of the selected integrity property between checked and unchecked.

The  (**Filter**) button is used in conjunction with the middle two panes of the window to filter the list of integrity properties. The left-hand pane of the two shows lists of the possible values for each of the attributes, while the right-hand pane shows the particular values of each attribute that the filter will use. Attribute values can be added to or removed from the filter by selecting them in

the appropriate pane and pressing the  (**Add to Filter**) or the  (**Remove from Filter**) button respectively. Pressing the  (**Filter**) button then causes the list of integrity properties to be filtered to show only those whose attributes match the selected attribute values. When no attributes are selected no filter is used so all integrity properties are shown. 

4.7 The Pretty Printer

The pretty printer transforms a specification from its input format to a pretty printed version of the specification. Typically this pretty printed version is used for documentation purposes. The output format for the pretty printer depends on the input format of the specification. If the input format is RTF the output format will also be RTF. If the input format is a mixture of \LaTeX commands and VDM++ specification the output format is a file which can be processed by \LaTeX . The main difference between the layout of the two different outputs produced by the pretty printer is that the one to Microsoft Word uses the ASCII version of VDM++ whereas the \LaTeX one uses a mathematical representation of VDM++ which is used in most VDM text books and articles.

The pretty printer can construct cross-referenced indexes and can also take test coverage information into account, both in the form of colouring of the parts of the specification that have not been covered and in the form of tables describing the percentage coverage of functions and operations.

If your input files are in RTF format, you can insert a table summarising the percentage test coverage by including the name of the class written in the `VDM_TC_TABLE` style at the desired position in the `.rtf` file, and test coverage colouring information is written by the pretty printer using the styles `VDM_COV` and `VDM_NCOV`. All three styles are included in the `VDM.dot` file from the Toolbox distribution.

The \LaTeX generator uses the `VDM++-VDMSL` macros together with the appropriate corresponding style file: `vpp.sty` for \LaTeX and `vdmsl-2e.sty` for $\text{\LaTeX}2_{\epsilon}$. These macros and style files are also supplied as part of the Toolbox distribution. Section 4.10 and Appendix B describe in detail how to set up the necessary \LaTeX environment for using the generated \LaTeX file.

The testing facilities are discussed further in Section 4.10.

The pretty printer can be accessed from either the GUI, the command line interface or the Emacs interface.

4.7.1 The graphical user interface

In order to invoke the pretty printer from the graphical user interface, first select the files you want the Toolbox to pretty print in the **Project View** of the **Manager**¹⁸,

¹⁸You can alternatively select classes in the **VDM View** of the **Manager** and the pretty printer is then applied to the set of files which contain the selected classes. This of course means that

then invoke the pretty printer by pressing the  (Pretty Printer) button.

Setting options

The pretty printer has a number of options which can be set in the **Pretty printer** pane of the **Project Options** window (see Figure 27). These options are:

Output index of definitions: Produces an index for definitions of functions, operations, types, instance variables and classes.
default: disabled.

Output index of definitions and uses: Produces an index for definitions of functions, operations, types, instance variables and classes, and for used occurrences of types, functions and operations. The Microsoft Word pretty printer is not able to take any uses of constructs into account and thus there is no difference between this option and the first option under Windows.
default: disabled.

Test coverage colouring: This enables highlighting in colour of those parts of the specification which have not been executed during testing. The coverage information is written to the test coverage file along with the standard test coverage information if this option is enabled.
default: disabled.

Note that only one of the options **Output index of definitions** and **Output index of definitions and uses** can be enabled at any time.

4.7.2 The command line interface

```
vppde -l [-nNr] specfile(s) ...
```

With the `-l` option `vppde` takes a VDM++ specification as its input and generates a pretty printed document. The format of this document depends on the input format. If the input format was RTF the name of the output file will be the same as the input file extended with `.rtf`. This generated file can stand alone and be

if a particular file contains more than one class definition and you select only some of those classes then the other classes in the same file are implicitly included.

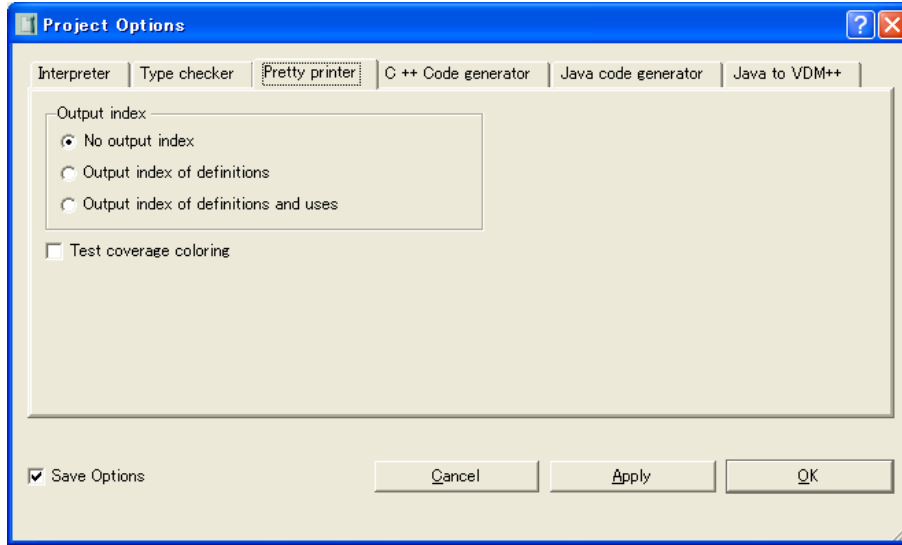


Figure 27: Setting Pretty Printer Options

taken into Microsoft Word directly. If the input format is a mix of \LaTeX and VDM++ specification the name of the output file will be the same as the input file extended with `.tex`. This generated file(s) can be included directly in any \LaTeX document.

The additional options which can be used with the pretty printer are:

- r Runs the pretty printer, inserting additional coverage information obtained from the test coverage file. For \LaTeX documents special macros are used in the specification to show which parts have or have not been exercised by the test suites. In the current version the test coverage file must be called `vdm.tc` and it must be placed in the working directory (which can be found using the `pwd` command). The test coverage file will have been generated by running the syntax analyser with its own `-R` option (see Section 4.3).

Section B describes in detail how to produce the test coverage report from the \LaTeX file produced by this command.

- n For RTF documents this option will mark all function/operation definitions with indexes. Inside the generated `.rtf` file you can then insert a table with all indexes by including the name of the class written in the `VDM.TC.TABLE` style at the desired position. For \LaTeX documents the option inserts \LaTeX macros around all definitions of functions, operations, types, states and

modules to be used to generate an index. Then an index can be produced using the `makeindex` utility.

`-N` For RTF documents this option is identical to the `-n` option. For \LaTeX documents this option works as `-n` but also inserts the macros around all applications of functions, operations, types and values.

4.7.3 The Emacs interface

In the Emacs interface there is only one command for the pretty printer. This command is, for historical reasons, called **latex** and, as for all other commands in the Emacs interface, it must be given at the command prompt.

latex (l) [-nNr] file

The pretty printer is invoked with `file`. If the \LaTeX format is used the VDM++ parts are typeset in the mathematical font with the VDM++- $\text{\texttt{VDM}\texttt{S}\texttt{L}}$ macros. If text parts exist, these and the VDM++ parts (VDM++- $\text{\texttt{VDM}\texttt{S}\texttt{L}}$ macros) are merged in the same order as in `file`. By using the `-n` or `-N` option indexes for defined and used occurrences will be generated (see Appendix B).


The option `-r` inserts coverage information collected in the test coverage file `vdm.tc`. For RTF documents the styles `VDM_COV` and `VDM_NCOV` must be defined in the input document. For \LaTeX documents this option inserts colours in the VDM++- $\text{\texttt{VDM}\texttt{S}\texttt{L}}$ macros such that all specification parts which have not been covered by the test suites are marked.

4.8 The VDM++ to C++ Code Generator

If you have a license for the VDM++ to C++ Code Generator you can have your specification automatically translated into C++ code by the Toolbox. Here we only explain how the code generator is invoked and what options it has; further details can be found in [7].

The C++ code generator can be accessed either from the GUI, from the command line version of the Toolbox, or from the Emacs interface.

4.8.1 The graphical user interface

In order to invoke the C++ code generator from the graphical user interface, first use the **Manager** to select the files or classes you want the Toolbox to translate, then invoke the translator by pressing the  (**Generate C++**) button. More than one file/class can be selected, in which case all of them are translated to C++.

The following options for the code generator can be set via the **C++ code generator** pane of the **Project Options** window which is shown in Figure 28:

Output position information Causes the code generator to generate code containing position information for run-time errors.
Default: **off**.

Check pre and post conditions Causes the code generator to generate code containing in-line checks of preconditions and postconditions of functions and of preconditions of operations.
Default: **on**.

4.8.2 The command line interface

```
vppde -c [-r] specfile, ...
```

With the **-c** option **vppde** generates code from the given **specfile(s)**. The specification is first parsed. Then, if no syntax errors are detected, the specification is type checked for possible well-formedness. Finally, if no type errors are detected, the specification is translated into a number of C++ files. The structure of the generated code and how to interface it are described in [7].

One additional option can be used with the VDM++ to C++ Code Generator:

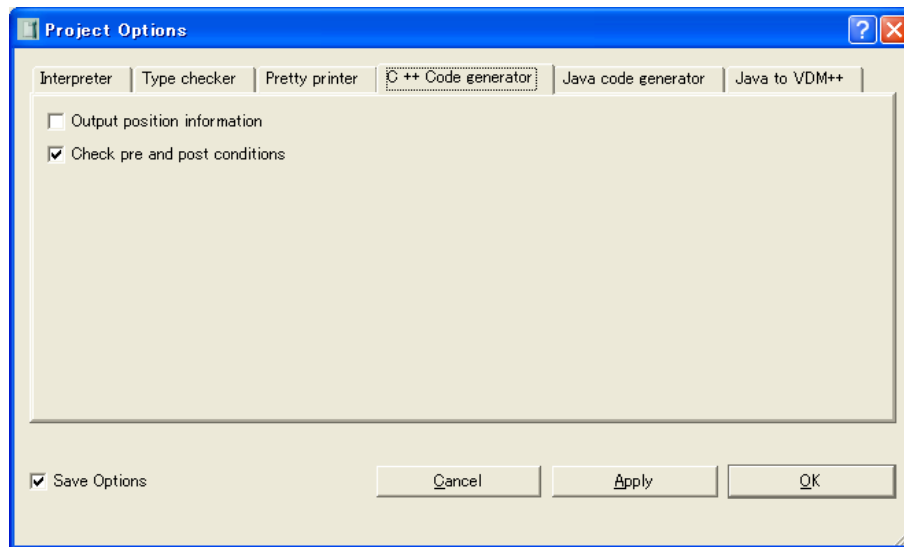


Figure 28: Setting Options for the C++ Code Generator

-r Includes run-time position information in the generated C++ code (for a detailed description see [7]).

4.8.3 The Emacs interface

In the Emacs interface there is only one command for the code generator. This command is called **codegen** and, as for all other commands in the Emacs interface, it must be given at the command prompt.

***codegen (cg) class [rti]**


This command generates C++ code for the class **class**. If the option **rti** is used run-time position information is included in the generated code.

4.9 The VDM++ to Java Code Generator

If you have a license for the VDM++ to Java Code Generator you can have your specification automatically translated into Java code by the Toolbox. Here we only explain how the code generator is invoked and what options it has; further details can be found in [8].

The Java code generator can be accessed either from the GUI, from the command line version of the Toolbox, or from the Emacs interface.

4.9.1 The graphical user interface

In order to invoke the Java code generator from the graphical user interface, first use the **Manager** to select the files or classes you want the Toolbox to translate, then invoke the translator by pressing the  (**Generate Java**) button. More than one file/class can be selected, in which case all of them are translated to Java.

The following options for the code generator can be set via the **Java code generator** pane of the **Project Options** window which is shown in Figure 29:

Generate only skeletons, except for types Causes the code generator to generate only skeleton classes, i.e. classes which contain full definitions of types, values and instance variables but empty definitions of functions and operations.

Default: **off**.

Generate only types Causes the code generator to generate only code corresponding to VDM++ type definitions, i.e. values, instance variables, functions and operations are ignored.

Default: **off**.

Generate integers as longs Causes the code generator to convert VDM++ integer values and variables to Java longs instead of Java integers.

Default: **off**.

Generate code with concurrency constructs Causes the code generator to generate code which includes support for concurrency.

Default: **on**.

Generate pre and post functions/operations Causes the code generator to generate code corresponding to preconditions, postconditions.

Default: **on**.

Check pre and post conditions Causes the code generator to generate code containing in-line checks of preconditions and postconditions of functions and of preconditions of operations.

Default: on.

Disable generate “vdm_” prefix Causes the code generator to generate code without the “vdm_” prefix in front of the user defined operations and functions. This can be convenient if for example one wishes to overload operations already defined in Java.

Selection of interface in case multiple inheritance is present in a VDM++ model it is necessary to reduce this for the Java code generation to single inheritance. This must be done by turning some of the classes to be translated to interfaces in Java. If that cannot be done it may be necessary to restructure the VDM++ model before it can be generated to Java.

Package for generated code can be used in case the user wishes the selected classes to be generated to a particular Java package.

In addition it is possible to select which classes in the specification are to be converted to Java interfaces, and to give the name for the package which the code generator creates to hold the Java code it generates.

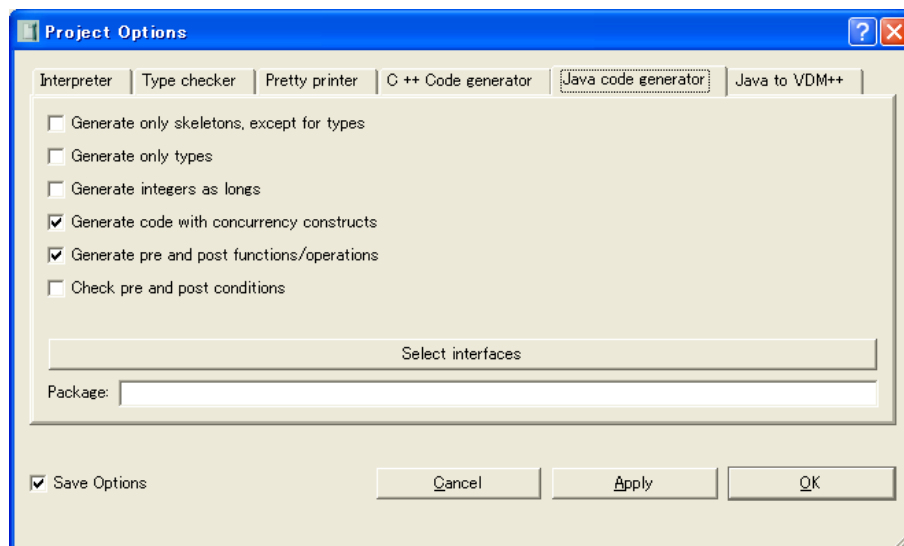


Figure 29: Setting Options for the Java Code Generator

4.9.2 The command line interface

```
vppde -j [options] specfile, ...
```

With the `-j` option `vppde` generates code from the given `specfile(s)`. The specification is first parsed. Then, if no syntax errors are detected, the specification is type checked for possible well-formedness. Finally, if no type errors are detected, the specification is translated into a number of Java files. The structure of the generated code and how to interface it are described in [8]. The different options available are also listed there.

4.9.3 The Emacs interface

In the Emacs interface there is only one command for the Java code generator. This command is called `javacg` and, as for all other commands in the Emacs interface, it must be given at the command prompt.

```
*javacg (jcg) class [options]
```

This command generates Java code for the class `class`.

4.10 Systematic Testing of VDM models

As part of its support for validation, the Toolbox provides a facility for testing VDM++ specifications, including test coverage measurement. Test coverage measurement helps you to see how well a given test suite covers the specification. This is done by collecting together in a special test coverage file information about which statements and expressions are evaluated during the execution of the test suite. The approach described here is a script-based one and is intended for application involving a large number of test cases. The approach described in Section 3 using the `tcov` command is better suited to a small number of test cases.

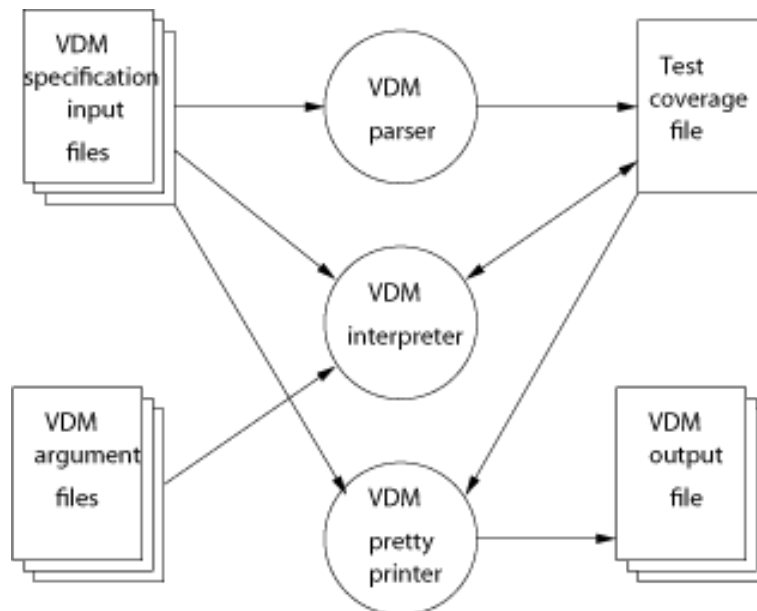


Figure 30: Systematic test of VDM models

There are three steps involved in producing a test coverage report (see Figure 30):

1. Prepare a *test coverage file*. VDM++ files in one of the input formats are first given to the VDM++ parser with a special option. This produces a test coverage file which contains information about the specification's structure but with none of the definitions covered yet.
2. The VDM++ interpreter is called with a number of arguments which are placed in small *argument files*. The interpreter is used with all the specification files and the test coverage file and then, in addition to returning the

result of evaluating the argument file, it updates the test coverage file with information about how often the different constructs have been exercised. The call of the interpreter is repeated with all the arguments in the test environment one wishes to take into account.

3. Finally the pretty printer is used with a special option which takes all the specification files and the test coverage file as input and produces a pretty printed version of the specification showing the detailed test coverage information from the test coverage file. Note that in the input VDM++ specification files only the textual parts which do not contain VDM++ definitions can be updated while this process is going on. If changes to the VDM++ parts happen the test coverage file does not know how to map its information back to the VDM++ specification files. Note also that on Windows the current version of the Toolbox requires that the test coverage file must be called `vdm.tc` and must be placed in the working directory.

4.10.1 Preparing the test coverage file

The generation of the test coverage information must be performed in a command prompt (under Windows this can be obtained by selecting the command prompt in the programs entry in the Windows setup; under Unix this is done in a normal shell). The parser must be invoked with the `-R` option. See Section 4.3.3 for details of the parameters to be used.

Example:

```
"vpphome/bin/vppde" -p -R vdm.tc Sorter.rtf DoSort.rtf ExplSort.rtf  
ImplSort.rtf MergeSort.rtf SortMachine.rtf
```

4.10.2 Updating the test coverage file

A test suite is normally structured into a hierarchy of directories where small argument files are placed in different categories depending upon what they are meant to test. In a development project you will wish to set up such a test environment and also make a small script file which will automate the testing process and even compare the actual results against expected results. In Appendix E there is an example of such a script file for both Windows and Unix. The test script must then call the command line interface of the Toolbox with the `-R` option. See Section 4.5.3 for details of the parameters to be used.

Example:

```
"vpphome/bin/vppde" -i -R vdm.tc -O dosort.res sort.arg  
Sorter.rtf DoSort.rtf ExplSort.rtf ImplSort.rtf MergeSort.rtf  
SortMachine.rtf
```

4.10.3 Producing the test coverage statistics

The result of running such a test suite can be displayed by using the pretty printer with appropriate options enabled. The pretty printer can be accessed from the graphical user interface, the command line interface and the Emacs interface. The important thing to remember is to use the option which enables coverage information to be incorporated. See Section 4.7 for details of the parameters to be used.

Example:

```
"vpphome/bin/vppde" -lr Sorter.rtf DoSort.rtf  
ExplSort.rtf ImplSort.rtf MergeSort.rtf SortMachine.rtf
```

In the pretty printed version of the files which are generated it is possible to see tables of the percentage coverage of the functions and operations in the specification. In addition detailed test coverage information showing how parts of functions and operations have been covered by the tests is available.

The **rtinfo** command, which can be input either through the command line interface or in the **Dialog** pane of the interpreter in the graphical user interface, displays test coverage information as explained below:

rtinfo vdm.tc

Before applying this command run-time information must have been collected in the test suite **vdm.tc**. The test suite is read and an overview of all functions and operations is displayed. For each element in this list, the number of times that element has been evaluated and the percentage coverage of its definition are shown. (This percentage is the number of covered expressions in the function/operation divided by the total number of expressions.) The total coverage of the test coverage file, which is the average of all the percentages in the list, is also shown.

4.10.4 Test coverage example using L^AT_EX

The way the different parts of the VDM++ input files differ with respect to test coverage depends on the input format. An example illustrating how to incorporate test coverage information into RTF files was presented in Section 3.9. The process for L^AT_EX files is quite different and is illustrated here for one of the sorting algorithms specified in the sorting example used throughout this manual. The specification can be found in the `vpphome/examples/sort/*.vpp` files.

In this small example `new DoSort().Sort([-12,5,45])` will be evaluated to show how well this small test covers the specification of the `DoSort` class.

The first step is to generate a test suite file using the parser:

```
prompt> vppde -p -R vdm.tc sorter.vpp dosort.vpp
Parsing "sorter.vpp" ... done
Parsing "dosort.vpp" ... done
prompt>
```

It is now possible to evaluate the argument file `sort.arg`, which contains a call to the `DoSort.Sort` VDM++ operation:

```
prompt> vppde -i -R vdm.tc sort.arg sorter.vpp dosort.vpp
Initializing specification ...
[ -12,5,45 ]
prompt>
```

When the interpreter is called with the `-R` option, it updates the test coverage file named `vdm.tc`.

Now the level of coverage of the `DoSort` class which has just been recorded in the file `vdm.tc` can be shown in the Toolbox. A table listing all the functions and operations in the specification together with the number of times they have been called and the percentage of coverage can be shown using the `rtinfo` command.

Each percentage is the number of covered expressions in the corresponding function/operation divided by the total number of expressions it contains. The total coverage of the whole test coverage file, which is the average of the percentages for the individual functions/operations, is also shown. Figure 31 illustrates this.

Now we call the pretty printer with an input file in L^AT_EX format:

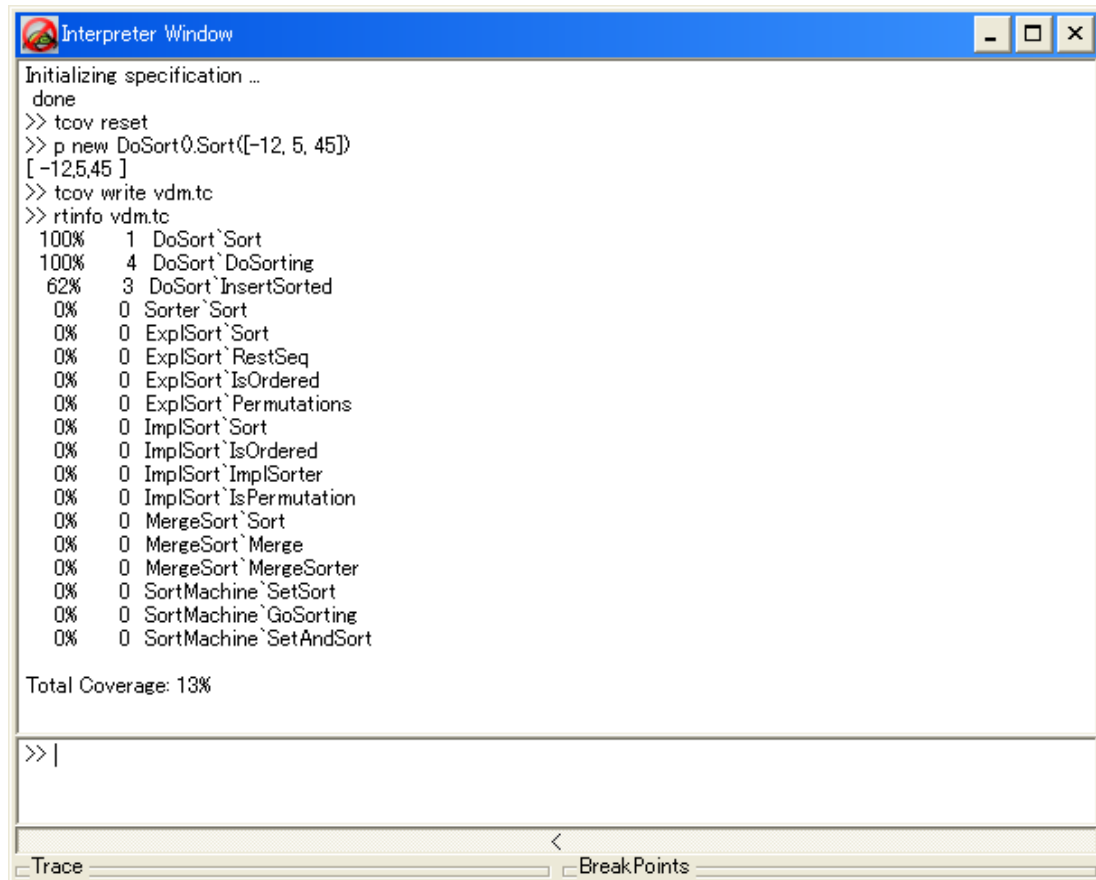


Figure 31: Showing test coverage information in the Toolbox

```

prompt> vppde -lr sorter.vpp dosort.vpp
Parsing "sorter.vpp" ... done
Parsing "dosort.vpp" ... done
Generating latex to file sorter.vpp.tex ... done
Generating latex to file dosort.vpp.tex ... done
prompt>

```

When the pretty printer is called with the `-r` option it will mark all those parts of the `DoSort`, `InsertSorted` and `Sort` operations and functions which have not been covered by our small test. Note that the option `-r` should only be used with $\text{\LaTeX}2\epsilon$ because older versions of \LaTeX do not support colours. After running \LaTeX on `sorter.vpp.tex` the result will appear as shown in Figure 32.


```

DoSort :  $\mathbb{R}^* \rightarrow \mathbb{R}^*$ 
DoSort (l)  $\triangleq$ 
  if l = []
  then []
  else let sorted = DoSort (tl l) in
    InsertSorted (hd l, sorted);

InsertSorted : PosReal  $\times$  PosReal*  $\rightarrow$  PosReal*
InsertSorted (i, l)  $\triangleq$ 
  cases true :
    (l = [])  $\rightarrow$  [i],
    (i  $\leq$  hd l)  $\rightarrow$  [i]  $\curvearrowright$  l,
    others  $\rightarrow$  [hd l]  $\curvearrowright$  InsertSorted (i, tl l)
  end

```

Figure 32: Test coverage of sorting example

Note that the **others** clause of the **cases** expression in **DoSort** ‘**InsertSorted**’ is not covered because **DoSort** ‘**Sort**’ was called with an already sorted sequence.

Appendix B gives a detailed description of how to combine VDM++ specifications with L^AT_EX.

Format of input file for L^AT_EX test coverage

This section describes how to include colouring of parts of the specification not covered by the test suites and tables showing coverage percentages in the L^AT_EX format. We illustrate this using the same sorting example used above. We discuss the generation of test coverage tables and the generation of colouring in turn.

Test coverage tables and the L^AT_EX test coverage environment

To insert tables that describe the number of calls and the coverage percentage of functions and operations the L^AT_EX environment **rtinfo** must be used. We first give an example illustrating the use of this environment, then we give part of a formal BNF-like definition of the usage.

In the sorting example the **rtinfo** environment is defined for the class **DoSort**. The **rtinfo** environment looks like:

Test Suite : vdm.tc
Class : DoSort

Name	#Calls	Coverage
DoSort'DoSorting	4	✓
DoSort'InsertSorted	3	62%
DoSort'Sort	1	✓
Total Coverage		79%

Figure 33: Example of a test coverage table

```
\begin{rtinfo}  
[TotalxCoverage]{vdm.tc}[DoSort]  
DoSorting  
InsertSorted  
Sort  
\end{rtinfo}
```

The first argument in the `rtinfo` environment (`TotalxCoverage` in the example) is optional. It is used to define the width of the column in the table which contains the function/operation names – the width will be the width of the argument. The second argument (`vdm.tc` in the example) is the name of the test coverage file. This argument is mandatory. The third (optional) argument (`DoSort` in the example) is the name of the class if the table is to be restricted to a particular class. If this argument is omitted all classes in the test coverage file are listed in the table.

Within the `rtinfo` environment, specific function and operation names can be written, in which case only those functions and operations are listed in the table. Otherwise all functions and operations are listed. In our example only the functions/operations `DoSort'DoSorting`, `DoSort'InsertSorted` and `DoSort'Sort` are listed.

The resulting table is shown in Figure 33.

The syntax of a test coverage environment is defined as follows:

$$\text{test coverage environment} = \text{'\begin{rtinfo}'}, \text{ test coverage section,} \\ \text{'\end{rtinfo}'};$$

```
test coverage section = [ long name ], test suite file, [ class name ],
                        [ function list ] ;
```

```
long name = '[', string, ']' ;
```

```
test suite file = '{', file identifier, '}' ;
```

```
file identifier = identifier, { '.', identifier } ;
```

```
class = '[', identifier, ']' ;
```

```
function list = { identifier } ;
```

Colouring

Colouring of the parts of a specification not covered by a test suite can only be shown if $\text{\LaTeX}2\epsilon$ is used. The \LaTeX file must be generated from the VDM++ specification to include test coverage colouring information, which is done by setting the **Enable test coverage colouring** option in the graphical user interface or by using the **-r** option in the command line interface and the **emacs** interface.

The following example shows the extra style files and definitions which are necessary to show the colouring:

```
\documentclass[dvips]{article}
\usepackage[dvips]{color}          <--- extra style
\usepackage{vpp}

\definecolor{covered}{rgb}{0,0,0}  %black  <--- extra
                                   %        definition
\definecolor{not-covered}{gray}{0.5} %gray   <--- extra
                                   %        definition

\begin{document}
...
\end{document}
```

In the generated L^AT_EX code the macros `\color{covered}` and `\color{not-covered}` are inserted in front of the specification parts that respectively have and have not been covered by the test suite. The `\definecolor` macros define that covered parts will be printed in black while the parts which were not covered will be grey. The result is shown in Figure 32 above.

The fact that the `others` clause of the `cases` expression in `DoSort` ‘`InsertSorted`’ is coloured grey in the L^AT_EX output means that this clause has not been covered (because `DoSort` ‘`Sort`’ has only been called with an already sorted sequence). On the basis of such information, the test suite can be improved to cover more of the specification.

For colour screens and colour printers red can also be used for not covered parts. In that case the definition macro should look like:

```
\definecolor{not-covered}{rgb}{1,0,0} %red
```

The `rtinfo` environment must also be included in order to use colouring, because the information used to generate the colouring is stored in the test coverage file.

References

- [1] CSK. *The Java to VDM++ User Manual*. CSK.
- [2] CSK. *The Rose-VDM++ Link*. CSK.
- [3] CSK. *VDM++ Installation Guide*. CSK.
- [4] CSK. *The VDM++ Language*. CSK.
- [5] CSK. *The VDM-SL Language*. CSK.
- [6] CSK. *VDM++ Sorting Algorithms*. CSK.
- [7] CSK. *The VDM++ to C++ Code Generator*. CSK.
- [8] CSK. The vdm++ to java code generator. Tech. rep.
- [9] CSK. *VDM Toolbox API*. CSK.
- [10] DICKINSON, I., AND LINES, K. Typesetting VDM-SL with VDM-SL macros. Tech. rep., National Physical Laboratory, Teddington, Middlessex, TW11 0LW, UK, July 1995.
- [11] FITZGERALD, J., AND LARSEN, P. G. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 1998. ISBN 0-521-62348-0.
- [12] FITZGERALD, J., LARSEN, P. G., MUKHERJEE, P., PLAT, N., AND VERHOEF, M. *Validated Designs for Object-oriented Systems*. Springer, New York, 2005.
- [13] P. G. LARSEN AND B. S. HANSEN AND H. BRUNN N. PLAT AND H. TOETENEL AND D. J. ANDREWS AND J. DAWES AND G. PARKIN AND OTHERS. Information technology — Programming languages, their environments and system software interfaces — Vienna Development Method — Specification Language — Part 1: Base language, December 1996.

Glossary

C++ Code Generator: The C++ Code Generator can automatically generate C++ code from your specification. You need a separate license for the C++ Code Generator in order to access the C++ Code Generator from the Toolbox.

Debugger: With the debugger you can explore the behaviour of your specification. The debugger can execute the specification and break at function and method applications. At any point in the execution you can explore the local or global state and local identifiers in your specification.

Dynamic Semantics: The dynamic semantics describes the meaning of a language. Thus, the dynamic semantics describes how the language behaves if it can be executed.

Emacs: Emacs is an ASCII editor.

GUI: Graphical User Interface.

Interpreter: The interpreter can interpret a specification according to the dynamic semantics of the language. That is, it can execute a program/specification.

Java Code Generator: The Java Code Generator can automatically generate Java code from your specification. You need a separate license for the Java Code Generator in order to access the Java Code Generator from the Toolbox.

L^AT_EX: is a generic typesetting system.

Pretty Printer: The pretty printer processes a file and produces a pretty printed version of the VDM++ parts in the input VDM++ file. The output format depends on the input format.

Project: A project is a collection of ASCII file names that make up a specification.

RTF: This is an acronym for “Rich Text Format” which is one of the formats which can be used with the Microsoft Word editor.

Semantics: describes the meaning of the language.

Specification: A specification is a VDM++ model of a system written in one or more files using potentially different input formats.

Static Semantics: The static semantics describes the relationships between the symbols of the language which must be obeyed in order for a syntactically correct specification to be well-formed (i.e. to have a consistent meaning). A well-formed specification is also called a type-correct specification.

Syntax: The syntax of a language describes how the symbol elements of the language (e.g. key words and identifiers) can be related. The syntax only describes how the symbols can be ordered in the language, not the meaning of the ordering.

Syntax Checker: A syntax checker verifies if the syntax of a specification is correct.

Test Coverage Information: Information about how many times each construct in the specification has been executed.

Test Coverage File: The test coverage file contains test coverage information.

Type Checker: The type checker checks the type correctness of a specification. A specification can be definitely or possibly type correct.

VDM: The Vienna Development Method.

VDM-SL: The formal specification language of the *Vienna Development Method*. VDM-SL is an ISO standard language [13].

VDM++: An object-oriented specification language that is an extension of ISO VDM-SL.

Well-formedness: A specification can be well-formed with respect to the syntax and the static semantics of a language.

A Information Resources on VDM Technology

This short appendix contains pointers to information resources that you may find helpful as you use VDM and the Toolbox.

Tutorial Books on VDM with close links to VDMTools

For Toolbox users, the following textbooks are certainly the most appropriate for tutorial purposes. They introduce concepts such as abstraction and analysis of formal models using a subset of the ISO Standard VDM-SL notation that is supported by the Toolbox and the object-oriented extension VDM++ respectively. In both books the ASCII notation, rather than the more arcane mathematical notation, is used and the presentation is guided by a series of examples based on industrial applications of the VDM technology. Most importantly, both books include many exercises which can be tackled using the Toolbox.

The two books are:

John Fitzgerald and Peter Gorm Larsen,
“Modelling Systems: Practical Tools and Techniques in Software Development”,
Cambridge University Press 1998,
ISBN 0-521-62348-0
<http://uk.cambridge.org/order/Webbook.asp?ISBN=0521623480>
(but since the book is now out of print you will need to find it second hand now)

and

John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat and Marcel Verhoef
“Validated Designs for Object-oriented Systems”
Springer, New York, 2005
ISBN 1-85233-881-4
<http://www.springer.com/east/home/generic/-search/results?SGWID=5-40109-22-33837368-0>

Supporting information (additional exercises, slides, web links, etc.) for both books can be found on the world-wide web at

<http://www.vdmbook.com/twiki/bin/view>.

World-wide Web Sites

The World-wide web contains a vast amount of information on formal methods in general and VDM in particular. Here it is only necessary to give a few sites because these contain links to the many others available.

The VDM Portal A web page containing basic information about VDM, including a bibliography, information on the VDM mailing list and a link to the VDM examples repository.

<http://www.vdmportal.org/twiki/bin/view/Main/WebHome/>

The VDM book site A web page with information about the VDM++ book and linked to various other VDM books.

<http://www.vdmbook.com/twiki/bin/view>

Formal Methods Europe The web pages for this organisation includes an applications database which is particularly interesting. This is an indexed database of applications of formal techniques, mostly in commercial or industrial contexts.

<http://www.fmeurope.org/>

The Formal Methods Archive A huge source of information on research and application in formal methods. Links to companies and organisations in formal methods, recommended introductory papers, and books.

<http://www.comlab.ox.ac.uk/archive/formal-methods/>

VDMTools information Web site A web page containing the information about VDMTools.

<http://www.vdmtools.jp/en/index.php>

Overture Web site A web page containing information about the open-source initiative called Overture on top of the Eclipse platform.

<http://www.overturetool.org/twiki/bin/view>

B Combining VDM++ and L^AT_EX

In this section a general description of how to construct a L^AT_EX document that contains VDM++ specification parts is given.

B.1 Format of a Specification File

If one wishes to use the L^AT_EX text processing system two different input formats can be used: one contains pure VDM++ ASCII specification, the other contains ASCII specification mixed with textual documentation. The latter type can be distinguished by the specification parts being enclosed within “`\begin{vdm_al}`” and “`\end{vdm_al}`”. The text-parts outside the specification blocks are ignored by the parser (but used by the pretty-printer). The “`\begin{vdm_al}`” cannot be placed arbitrarily in a specification, but only in front of the keywords `class`, `instance variables`, `functions`, `operations`, `values` and `types` and in front of definitions of `functions`, `operations`, `types` and `values`. This means for instance that a text-part cannot be inserted in the middle of the body of a function. The file:

```
vdmhome/examples/sort/mergesort.vpp
```

gives an example of how specification and text can be mixed.

B.2 Setting up a L^AT_EX Document

The general combination of VDM++ and L^AT_EX can be used both with the original L^AT_EX program and with the newer L^AT_EX2 ϵ , whereas the incorporation of test coverage is possible only when using L^AT_EX2 ϵ because some features, such as colouring within documents, are only available with L^AT_EX2 ϵ .

The following example of L^AT_EX code shows the L^AT_EX style files that must be included to generate a general L^AT_EX document which includes VDM++ parts:

The `vpp` style file is included in the Toolbox distribution.

The L^AT_EX heading can either be inserted into one of the VDM++ specification files or it can be put in an isolated file that includes the VDM++ specification files. In either case the specification files must be translated into L^AT_EX files by the

```
\documentstyle[vpp]{article}

\begin{document}
...
\end{document}
```

Figure 34: Example of an original \LaTeX Document

```
\documentclass{article}
\usepackage{vpp}

\begin{document}
...
\end{document}
```

Figure 35: Example of a $\text{\LaTeX}2\epsilon$ Document

Toolbox, either by pressing the **Pretty Print** button in the graphical user interface, by using the `latex` command in the **Emacs** interface, or by using the `-l` option in a command line call to `vppde`. For the sorting example the heading has been inserted into the `sort.tex` file.

Line numbering

All definitions in the generated \LaTeX are by default given definition numbers and line numbers. These numbers can be suppressed using the following commands:

```
\nolinenumbering
\setindent{outer}{\parindent}
\setindent{inner}{0.0em}
```

For further information see [\[10\]](#).

Indexes

Macros to generate index numbers can be generated as part of the \LaTeX pretty printing output. The index numbers refer to pages in the final \LaTeX document.

Indexes can be generated at two levels: indexes for all definitions of classes, functions, operations, types and instance variables, and indexes for all uses of functions, types, and classes. Index macros for definitions only are generated in the graphical interface by enabling the **Output index of definitions** option, and in the command line and emacs interfaces by using the `-n` option. Index macros for both definitions and uses are generated in the graphical interface by enabling the **Output index of definitions and uses** option, and in the command line and emacs interfaces by using the `-N` option.

The index numbers are automatically inserted into different \LaTeX macros to clarify which kind of construct an index refers to. The macros for definitions are:

- `InstVarDef` indicating where an instance variable is defined.
- `TypeDef` indicating where a type is defined.
- `FuncDef` indicating where a function or operation is defined.
- `ClassDef` indicating where a class is defined.

The macros for uses are:

- `TypeOcc` indicating where a type is used.
- `FuncOcc` indicating where a function is used. Only the use of functions that are explicitly defined in the document will be indexed.
- `ClassOcc` indicating where a class is used.

These \LaTeX macros must be defined at the beginning of the \LaTeX document in order to use indexing. An example of this is:

```
\newcommand{\InstVarDef}[1]{\bf #1}
\newcommand{\TypeDef}[1]{\bf #1}
\newcommand{\TypeOcc}[1]{\it #1}
\newcommand{\FuncDef}[1]{\bf #1}
\newcommand{\FuncOcc}[1]{#1}
\newcommand{\ClassDef}[1]{\sf #1}
\newcommand{\ClassOcc}[1]{#1}
```

Four additional parts have to be included in the \LaTeX document in which you want to include the index:

1. Include the `makeidx` style option in `\documentstyle` if using the original \LaTeX , or include it as a package if using $\text{\LaTeX}2\epsilon$.
2. Include `\makeindex` in the preamble of the document.
3. Define the macros `InstVarDef`, `TypeDef`,
4. Include `\printindex` at the position where you would like to have the index included in the document.


Unsupported constructs

There is one syntactical construct that is not supported currently by the \LaTeX pretty printer: at present it is not possible to typeset comments. These will simply be ignored by the \LaTeX pretty printer. It is recommended to mix specifications and text instead of using VDM++ comments.

C Setting up your VDMTools Environment

A number of environment variables and other options for the Toolbox as a whole can be set according to your personal preferences. These are described below.

C.1 Interface Options

The following interface options can be set through the **Edit and Print** pane of the **Tool Options** window which is opened by selecting the **Tool Options** () item from the **Project** toolbar/menu and which is shown in Figure 36:

External Editor: Options to define the external editor that can be invoked from within the Toolbox.

Editor Name: The name of the external editor to use. Unix default value: emacsclient. Windows default value: notepad. Note that it is not yet possible to use Microsoft Word here directly.

Format to load single file: default value: +%1 %f

Format to load multiple files: default value: %f

Print command: default: 1pr **Note:** on the Windows platform it is not possible to use this at all because the pipe and print icons are not present at all.

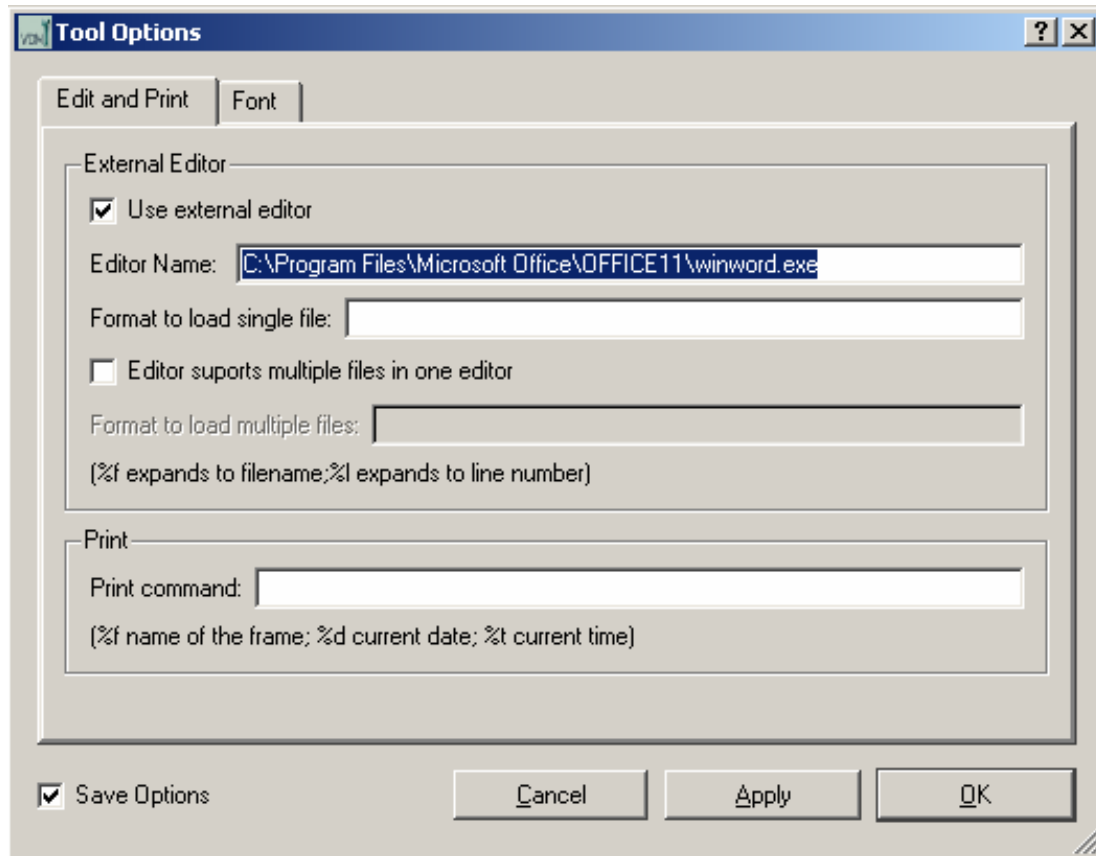



Figure 36: Setting Editor and Print Options

C.2 Multilingual Support

The Toolbox supports a range of fonts which allow a corresponding range of scripts to be used both in the specification itself (i.e. in the names of the identifiers used in the specification) and in the more general text accompanying the specification. The relevant language support should first be installed appropriately at the level of the basic operating system (Windows or Unix) within which the Toolbox is running, then the Toolbox can be configured to use this through the **Font** pane of the **Tool Options** window (see Figure 37) which is invoked via the **Tool Options** item () on the **Project** toolbar/menu. Press the **Select Font** button in the **Font** pane of the **Tool Options** window to open a browser containing a list of available fonts (at the level of the operating system) and select the font you want. Finally, select the appropriate encoding from the **Text Encoding** menu

in the same pane.

On the font pane it is also possible to select whether one wish to have **syntax coloring** and wish to use **auto syntax checking** (both are selected by default). When **syntax coloring** is selected the VDM keywords will be highlighted in the **source window**. When **auto syntax checking** is selected files will automatically be syntax checked when they are saved in a newer version to the file system.

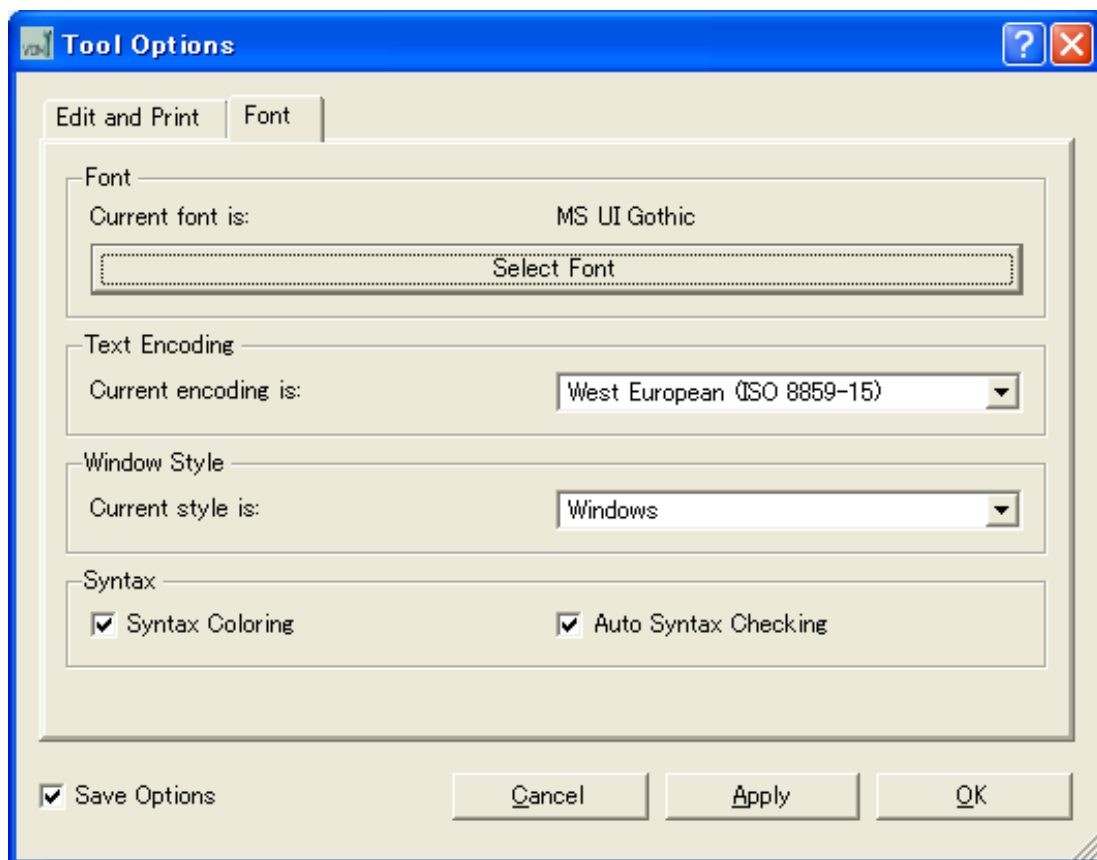


Figure 37: Setting Font Options

Note that the range of scripts available in the Toolbox is limited to those which are supported by the general Qt interface. The possible selections offered when setting the text encoding represent those currently supported.

C.3 UML Link support

At this pane it is possible to select the desired interface to UML and the external UML tool. At the moment there is support to JUDE:

`http://jude.change-vision.com/jude-web/index.html`

and Enterprise Architect:

`http://www.sparxsystems.com.au/products/ea/downloads.html`

Both of these links are made using the XMI exchange standard between UML tools. Here is is also possible to state the preferred file type at the VDM level as well as the template one wish to use for such files.

D The Emacs Interface

The Emacs Interface is only supported on the different Unix platforms where the Toolbox is available. The file `vppde.el` contains the Emacs macros which enable one to get access to the Toolbox functionality directly from Emacs. The guided tour from Section 3 can be followed by using the file `mergesort.init` instead of the `mergesort-init.rtf` file.

After installing the Toolbox and updating your `.emacs` file as described in the document [3] you can run the Emacs editor.

From the Emacs editor call the VDM++ Development Environment (`vppde`) by typing `M-x vppde` (Press the meta-key and the x-key and give `vppde` as the argument). This can be done from any Emacs buffer. You will now be prompted for a file name which must contain a VDM++ specification. Type `mergesort.vpp` and press RETURN.

When `vppde` is started it will parse the input file and if some syntax errors are discovered the Emacs window will be split into two. One half will contain the `mergesort.vpp` file (from now on called the specification window) and the other will be the command window for `vppde` (from now on called the command window). The syntax errors are indicated directly in the specification window by the marker `=>`.

This is similar to the explanation given in Section 3 and you can continue the guided tour following that example. By typing `help` on the command line in the command window you will also be able to get an overview of how the different features can be accessed.

E Test Scripts for the Sorting Example

The test scripts shown here consist of two scripts:

- The specific test script that tests a single argument. It takes the name of an argument file as parameter.
- The top level test script that loops over a number of arguments files. These argument files can be organised in an entire directory hierarchy. For each argument file it calls the specific test script with that file name.

The test scripts depend on the platform and thus this appendix is divided into different parts depending upon the platform. More advanced test scripts can be made. Those presented here are simple ones intended to demonstrate the basic approach.

E.1 The Windows/DOS Platform

The top level test script is called `vdmlloop.bat` and can be executed in a DOS command prompt. This file is available in the Toolbox distribution in the `examples/sort/test` directory. It looks like:

```
@echo off
rem -- Runs a collection of VDM++ test examples for the
rem -- sorting example
set SPEC1=..\DoSort.rtf ..\SortMachine.rtf ..\ExplSort.rtf
set SPEC2=..\ImplSort.rtf ..\Sorter.rtf ..\MergeSort.rtf

vppde -p -R vdm.tc %SPEC1% %SPEC2%
for /R %%f in (*.arg) do call vdmtest "%%f"
```

The test script, which takes one argument file, is called `vdmtest.bat`. This file is also available in the Toolbox distribution. It looks like:

```
@echo off
rem -- Runs a VDM test example for one argument file
```

```
rem -- Output the argument to stdout (for redirect) and
rem -- "con" (for user feedback)
echo VDM Test: '%1' > con
echo VDM Test: '%1'
set SPEC1=..\DoSort.rtf ..\SortMachine.rtf ..\ExplSort.rtf
set SPEC2= ..\ImplSort.rtf ..\Sorter.rtf ..\MergeSort.rtf

vppde -i -R vdm.tc -O %1.res %1 %SPEC1% %SPEC2%

rem -- Check for difference between result of execution and
rem -- expected result.
fc /w %1.res %1.exp

:end
```

The reason for defining both SPEC1 and SPEC2 instead of simply one variable is to avoid one very long line.

E.2 The UNIX Platforms

The top level test script is called `vdmloop` and can be executed in a normal shell. This file is available in the Toolbox distribution in the `examples/sort/test` directory. It looks like:

```
#!/bin/sh

## Runs a collection of VDM++ test examples for the sorting example.
SPEC="../dosort.vpp ../implsort.vpp ../sorter.vpp ../explsort.vpp \
      ../mergesort.vpp ../sortmachine.vpp"

## Generate the test coverage file vdm.tc
vppde -p -R vdm.tc $SPEC

## Find all argument files and run them on the specification.
find . -type f -name \*.arg -exec vdmtest {} \;
```

The test script, which takes one argument file, is called `vdmtest.bat`. This file is also available in the Toolbox distribution. It looks like:

```
#!/bin/sh

## Runs a VDM test example for one argument file.

## Output the argument to stdout (for redirect) and
## "/dev/tty" (for user feedback)
echo "VDM Test: '$1'" > /dev/tty
echo "VDM Test: '$1'"

SPEC="../dosort.vpp ../implsort.vpp ../sorter.vpp ../explsort.vpp \
      ../mergesort.vpp ../sortmachine.vpp"

## Run the specification with argument while collecting
## test coverage information, and write the result to an
## output file.
vppde -i -R vdm.tc -O $1.res $1 $SPEC

## Check for difference between result of execution
## and expected result.
diff -w $1.res $1.exp
if test $? = 0 ; then
    echo "SUCCESS: Result equals expected result" > /dev/tty
    echo "SUCCESS: Result equals expected result"
else
    echo "FAILURE: Result differs from expected result" > /dev/tty
    echo "FAILURE: Result differs from expected result"
fi
```

F Troubleshooting Problems with Microsoft Word

Breakpoints on lines

The `VDM.dot` Word template file included in the **VDMTools** distribution includes a special macro enabling the setting of breakpoints on lines inside functions and operations. However this has only been included from version v9.0.6 of the VDM++ Toolbox. The macro can be activated by pressing **Control-Alt-spacebar** when the cursor is placed on the desired line in an RTF file which is included in the current project.

1. Before activating the macro the Toolbox interpreter must be initialised by pressing the **Init** button.
2. Maybe you have forgotten to move the `VDM.dot` file into the template directory used by Word; usually this is

`C:\Program Files\Microsoft Office\Templates`

3. The document you are using may not have `VDM.dot` as its template. This can be checked using the **Files->Properties** facility inside Microsoft Word. If you are not using `VDM.dot` make sure that the macro is copied over to the template you wish to use instead.

Finding the test coverage file

If the test coverage file cannot be included correctly it could be because you are using files which are placed on a Unix server which is accessed from Windows via Samba. In this situation it is currently not possible to correctly distinguish between upper and lower case letters used in file names, so you should make sure that all file names are using lower case.

G Format for Priority File

Priority files should have the following format:

priority file = priority entry { ‘,’ , **priority entry** } ‘,’ ;

priority entry = class name ‘:’ numeral ;

numeral = **digit**, { **digit** } ;

digit = ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’ ;

Index

- .vppde file, 47
- .emacs file, 107
- \$\$, 62, 63, 67, 70
- vppde, 107
- vppde
 - command line options, 67
- auto syntax checking, 105
- backtrace command, 69
- break command, 60, 61, 69
- Breakpoints, 22, 61, 70
 - Deleting, 24, 58, 60, 62
 - Disabling, 24, 58, 62, 70
 - Enabling, 24, 58, 62, 70
 - Ignoring, 23, 60, 68
 - Removing, 71
 - Setting, 22, 60, 61, 69
 - Setting in Microsoft Word, 61
- C++ Code Generation, 34
- C++ Files, 81
- Call stack, 23
- Call stack, 24, 58
- Class View, 10
- classes command, 45
- Code Generation, ⇒
 - C++ Code Generation,
 - Java Code Generation
- codegen command, 82
- Command Line Interface, 44
 - C++ code generator, 81
 - Debugger, 67
 - Initialisation File, 46
 - Interpreter, 67
 - Java code generator, 85
 - Pretty Printer, 78
 - Starting, 44
 - Syntax Checker, 49
 - Type Checker, 55
- cont command, 69
- cquit command, 46
- create command, 60, 61, 69
- curthread command, 60, 61, 69, 71
- debug command, 23, 60, 61, 70
- Definite Well-formedness, 52, 53, 55
- delete command, 60, 62, 70
- destroy command, 60, 62, 70
- dir command, 46
- disable command, 62, 70
- Dynamic Type Checking, ⇒
 - Type Checking,
 - Dynamic
- Emacs Interface, 107
- enable command, 62, 70
- Error List, 10, 13, 48, 53
- External Editor, 14, 44
- finish command, 70
- first command, 50, 56
- Functions
 - Implicit, 58
 - Precondition, 21
- functions command, 45
- Graphical User Interface, 36
 - Starting, 8, 36
- Help, 8, 44
- help command, 46
- info command, 46
- init command, 62, 70
- Input
 - Creating, 7
 - Formats, 1
 - Multilingual, 1

- instvars command, 45
- Integrity Properties, 27
- Interpreter, 19
 - Commands, 59
 - Initialising, 20, 59, 60
 - Stopping, 62, 63
- Interpreter Window, 10, 19, 58
- Invariant Checking, 26
- Java Code Generation, 34
- Java Files, 85
- javacg command, 85
- last command, 50, 56
- latex command, 80
- License, 34, 81, 83
- Log Window, 10
- Manager, 10, 37
- next command, 51, 56
- Non-executable Constructs, 22
- objects command, 60, 62, 70
- operations command, 45
- option command, 72
- Options
 - C++ Code Generator, 82
 - Editor and Print, 104
 - Interpreter, 26
 - Java Code Generator, 84
 - Pretty Printer, 79
 - Project, 40
 - Tool, 40
 - Type Checker, 54
- Options:Interpreter, 65
- pop command, 63, 70
- popd command, 62, 70
- Possible Well-formedness, 52, 53, 81, 85
- Postcondition Checking, 26, 65, 68
- Precondition Checking, 26, 65, 68
- Precondition Functions, \Rightarrow
 - Functions,
 - Precondition
- Pretty Printing, 33
 - Comments, 102
 - Example, 99
 - Indexes, 100
 - Setting up L^AT_EX document, 99
- previous command, 51, 56
- print command, 23, 60, 62, 71
- Priority File
 - Format of, 112
- priorityfile command, 63, 71
- Project, 37
 - Adding files, 11
- Project View, 10
- push command, 63, 71
- pwd command, 33, 46
- quit command, 46
- read command, 50
- remove command, 71
- rtinfo command, 32, 88
- script command, 46
- selthread command, 60, 63
- set command, 56
- set full command, 56
- singlestep command, 72
- Source Window, 10
- Status Information, 15
- step command, 72
- stepin command, 73
- Syntax Errors, 13
- Syntax Checking, 15
- syntax coloring, 105
- Syntax Errors, 48, 107
 - Correcting, 13
 - Format, 49
- SyntaxChecking, 12
- system command, 46

- tcov write command, [32](#)
- tcov command, [64](#), [73](#)
- tcov read command, [64](#), [73](#)
- tcov reset command, [32](#), [64](#), [73](#)
- tcov write command, [64](#), [73](#)
- Test Coverage, [31](#)
- Test Coverage, [86](#), [89](#)
 - Environments, [92](#)
 - Examples, [89](#)
 - File, [32](#), [79](#), [86](#)
 - Test suite, [31](#), [86](#)
- Test Suite, \Rightarrow
 - Test Coverage,
 - Test suite
- Testing, \Rightarrow
 - Test Coverage
- threads command, [60](#), [63](#), [73](#)
- Type Checking, [15](#), [52](#)
 - Dynamic, [24](#), [68](#)
- Type Correctness, \Rightarrow
 - Possible Well-formedness,
 - Definite Well-formedness
- Type Errors, [53](#)
- typecheck command, [56](#)
- types command, [45](#)

- unset command, [73](#)
- unset full command, [57](#)

- values command, [45](#)
- VDM Standard, [1](#)
- VDM.dot file, [7](#), [61](#), [77](#), [111](#)
- vdmgde command, [8](#), [36](#)
- vpp.sty file, [77](#)

- Well-formedness, \Rightarrow
 - Possible Well-formedness,
 - Definite Well-formedness