Overture VDM-10 Tool Support: User Guide

by

Peter Gorm Larsen, Kenneth Lausdahl, Augusto Ribeiro and Sune Wolff
Engineering College of Aarhus
Dalgas Avenue 2, DK-8000 Århus C, Denmark

Nick Battle
Fujitsu Services
Lovelace Road, Bracknell,
Berkshire. RG12 8SN, UK

Overture – Open-source Tools for Formal Modelling

Document history

Month	Year	Version	Version of Overture.exe
January	2010		0.1.5
March	2010		0.2
May	2010	1	0.2
February	2011	2	1.0.0
June	2011	3	1.0.1

Contents

ABSTRACT

This document is the user manual for the Overture Integrated Development Environment (IDE) for VDM. It serves as a reference for anybody wishing to make use of the tool with one of the VDM dialects (VDM-SL, VDM++ or VDM-RT). Overture tool support is build on top of the Eclipse platform. The objective of the Overture initiative is to create and support an open source platform that can be used for both experimentation with new VDM dialects, as well as new features for analysing VDM models in different ways. The tool is entirely open source, so anybody can join the development team and influence future developments. The long term goal is to ensure that stable versions of the tool suite can be used for large scale industrial applications of VDM technology.

Chapter 1

Introduction

The Vienna Development Method (VDM) is one of the longest established model-oriented formal methods for the development of computer-based systems and software [Bjørner&78a, Jones90, Fitzgerald&08a]. It consists of a group of mathematically well-founded languages for expressing system models during early design stages, before expensive implementation commitments are made. The construction and analysis of a model using VDM helps to identify areas of incompleteness or ambiguity in informal system specifications, and provides some level of confidence that a valid implementation will have key properties, especially those of safety or security. VDM has a strong record of industrial application, in many cases by practitioners who were not specialists in the underlying formalism or logic [Larsen&95, Clement&99, Kurita&09]. Experience with the method suggests that the effort expended on formal modelling and analysis can be recovered in reduced rework costs arising from design errors.

VDM models are expressed in a specification language (VDM-SL) which supports the description of data and functionality [ISOVDM96, Fitzgerald&98, Fitzgerald&09]. Data are defined by means of types built using constructors that define structured data and collections such as sets, sequences and mappings from basic values such as Booleans and natural numbers. These types are very abstract, allowing you to add any relevant constraints using data type invariants. Functionality is defined in terms of operations over these data types. Operations can be defined implicitly by preconditions and postconditions that characterize their behavior, or explicitly by means of specific algorithms. An extension of VDM-SL, called VDM++, supports object-oriented structuring of models and permits direct modelling of concurrency [Fitzgerald&05]. An additional extension to VDM++, called VDM Real Time (VDM-RT [1]), includes support for discrete time models [Mukherjee&00, Verhoef&06]. All three VDM dialects are supported by Overture.

Since VDM modelling languages have a formal mathematical semantics, a wide range of analyses can be performed on models, both to check internal consistency and to confirm that models have emergent properties. Analyses may be performed by inspection, static analysis, testing or mathematical proof. To assist in this process, Overture offers tool support for building models in collaboration with other modelling tools, to execute and test models and to carry out different forms of static analysis [Larsen&10]. It can be seen as an open source version of the commercial

---

[1] Formerly called VDM In a Constrained Environment (VICE))

tool called VDMTools [Elmstrøm&94, Larsen01, Fitzgerald&08b] although features to generate executable code in high-level programming languages are not yet available in Overture.

This guide explains how to use the Overture IDE for developing models for different VDM dialects. It starts with an explanation of how to get hold of the software in Chapter 2. This is followed in Chapter 3 with an introduction to the Eclipse workspace terminology. Chapter 4 explains how projects are managed in the Overture IDE. Chapter 5 covers the features for creating and editing VDM models. This is followed in Chapter 6 with an explanation of the interpretation and debugging capabilities in Overture. Chapter 7 illustrates how test coverage information can be gathered when models are interpreted. Chapter 8 shows how models with test coverage information can be written as LaTeX and automatically converted to PDF format. Chapters 9 to 13 cover various VDM specific features: Chapter 9 explains the notion of proof obligations and their support in Overture; Chapter 10 explains combinatorial testing and the automation support for that; Chapter 11 explains the support for mapping between object-oriented VDM models and UML models; Chapter 12 shows how a VDM++ project can be converted into a new VDM-RT project; Chapter 13 shows how to analyse and display execution logs from VDM-RT models; and Chapter 16 gives an overview of the features of Overture which are also available from a command-line interface. Appendix A provides a list of all the standard templates built into Overture. Appendixes B to G give complete lists of possible errors, warnings and proof obligation categories. Finally, Appendix H is an index of significant terms used in the user manual.

Chapter 2

Getting Hold of the Software

Overture is an open source VDM tool, developed by a community of volunteers, and built on top of the Eclipse platform. The project is hosted on SourceForge. The best way to obtain Overture is to download a special version of Eclipse with the Overture functionality already pre-installed. If you go to:

> `http://sourceforge.net/projects/overture`

you can use the *Download Now* button to automatically download pre-installed versions of Overture for your operating system. Supported systems are: Windows, Linux and Mac. Simply extract the zip file downloaded and run the Overture executable file at the top level to start the tool. Note that in order to be able to execute Overture you need to have Java Runtime Environment (minimum version 1.5) installed on your computer. When you start Overture for the first time, it will request a workspace location. We recommend that you choose the default location proposed by Overture and tick the box for "Use this as the default and do not ask again". A welcome screen will introduce you to the overall mission of the Overture open source initiative the first time that you run the tool and provide you with a number of interesting pointers of additional material (see Figure 2.1). You can always get back to this page using *Help → Welcome*.

Large libraries of sample VDM-SL, VDM++ and VDM-RT models are available and can be downloaded from SourceForge under the `files/Examples` section using the URL[1]:

> `https://sourceforge.net/projects/overture/files/Examples/`

Such existing projects can be imported into Overture as described in section 4.1.

---

[1]The library files are intended to be used with Eclipse, but can also be opened with file compression programs like `Winrar` on Windows
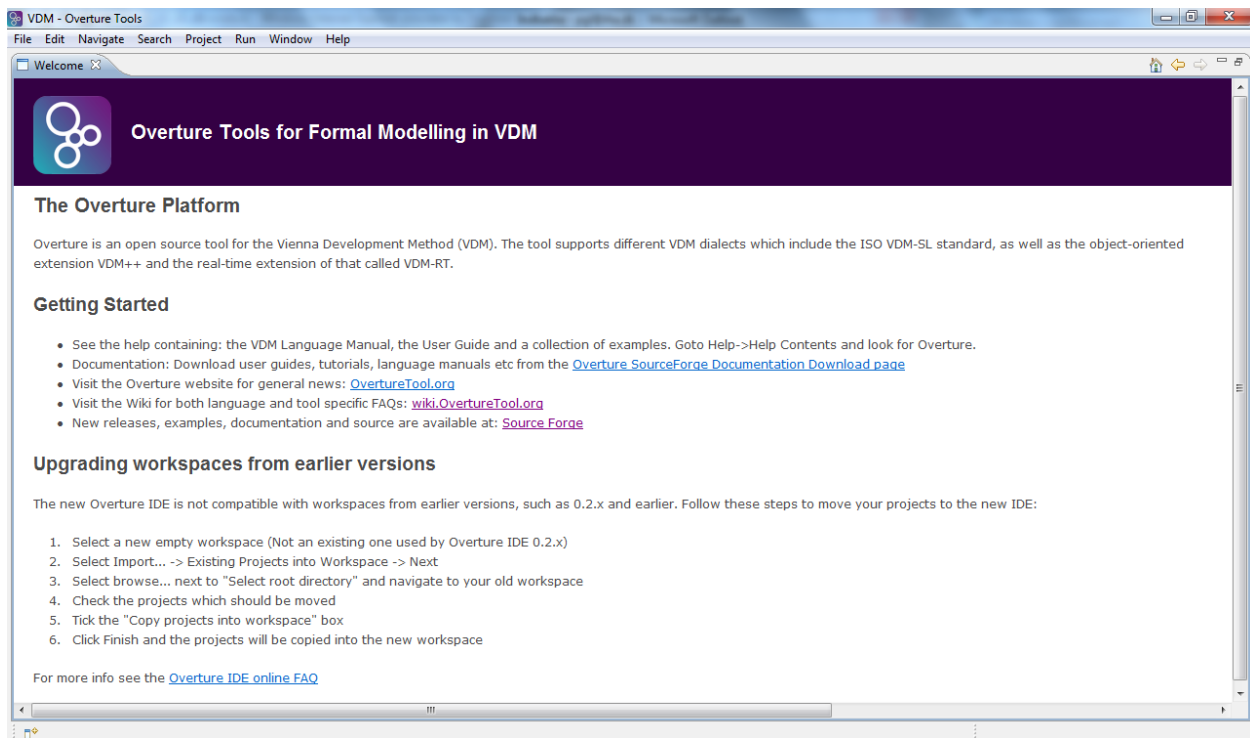
Figure 2.1: The Overture Welcome Screen

Chapter 3

Using the VDM Perspective

3.1 Understanding Eclipse Terminology

Eclipse is an open source platform based around a *workbench* that provides a common look and feel to a large collection of extension products. If you are familiar with one Eclipse product, you will generally find it easy to start using other products that use the same workbench. The Eclipse workbench consists of several panels known as *views*, such as those shown in Figure 3.1. A particular arrangement of views is called a *perspective*, for example Figure 3.1 shows the standard VDM perspective. This consists of a set of views for managing Overture projects and viewing and editing files in a project. Different perspectives are available in Overture as will be described later, but for the moment think of a perspective as a useful composition of views for conducting a particular task.

The *VDM Explorer* view lets you create, select, and delete Overture projects and navigate between the files in these projects, as well as adding new files to existing projects. A new VDM project is created choosing the *File → New → Project* option which results in the dialog shown in Figure 3.2. Select the desired VDM dialect and press *Next*. Finally, the project needs to be given a name. Click `Finish` to create the project. Depending upon the dialect of VDM used in a given project, a corresponding Overture *Editor* view will be available to edit the files of your new project. Dialect editors are sensitive to the keywords used in each particular dialect, and simplify the task of working on the specification.

The *Outline* view, on the right hand side of Figure 3.1, presents an outline of the file selected in the editor. The outline shows all VDM definitions, such as state definitions, values, types, functions and operations. The type of each definition is also shown in the view and the colour of the icons in front of the names indicates the accessibility of each definition. Red is used for private definitions, yellow for protected definitions and green for public definitions. Triangles are used for type definitions, small squares are used for values, state components and instance variables, functions and operations are represented by larger circles and squares, permission predicates are shown with small lock symbols and traces are shown with a "T". Functions have a small "F" superscript over the icons and static definitions have a small "S" superscript. Record types have a small arrow in front of the icon, and if that is clicked the fields of the record can be seen.
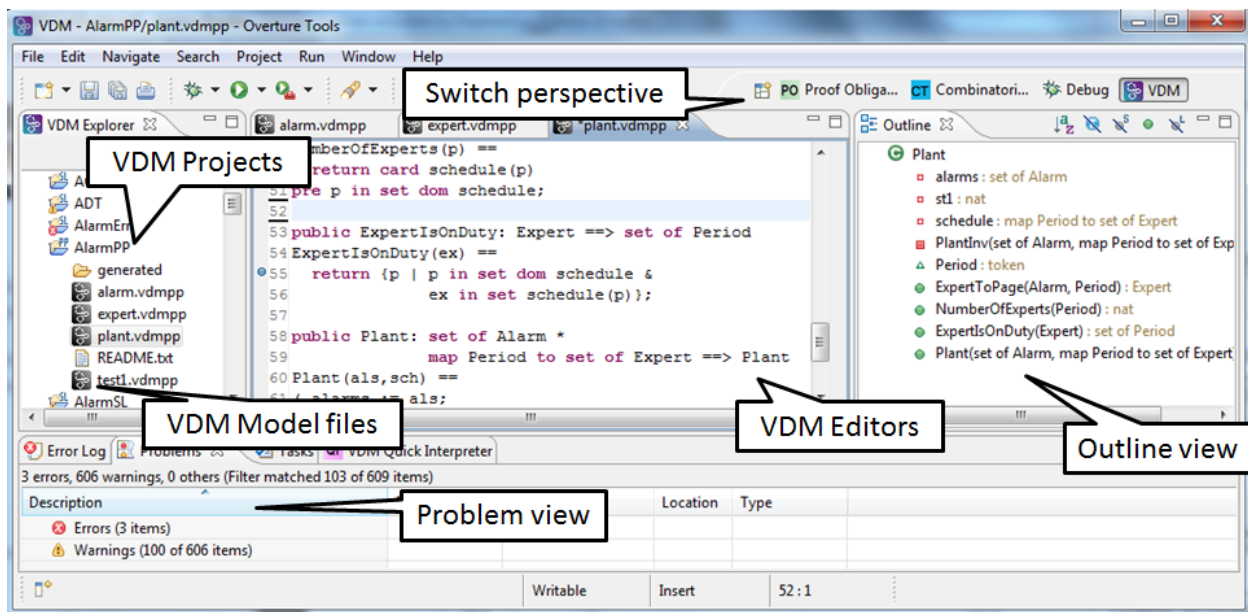
Figure 3.1: The VDM Perspective
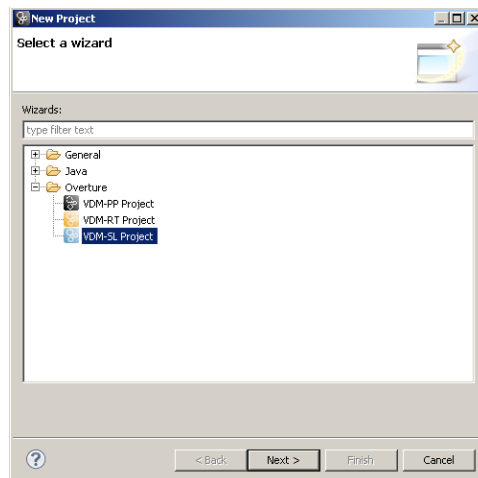
Figure 3.2: Creating a New VDM Project

Figure 3.3 illustrates the different outline icons. At the top of the view there are buttons to filter what is displayed, for instance it is possible to hide non-public members.

Clicking on the name of a definition in the outline will navigate to the definition and highlight the name in the Editor view.

The *Problems* view at the bottom of Figure 3.1 displays information messages about the projects you are working on, such as warnings and syntax or type checking errors.

Figure 3.3: Icons in the Outline View

The *VDM Quick Interpreter* view has a small command-line at the bottom where a plain VDM expression (not depending upon the definitions in the VDM model you are working with but for that you can use the "Console" launch mode explained in Section 6.1) can be entered. When return is pressed, the expression will be evaluated and the result shown above the command-line.

Most of the other features of the workbench, such as the menus and toolbars, are similar to other Eclipse applications, with the exception of a special menu with Overture specific functionality.

3.2 Additional Eclipse Features Applicable in Overture

3.2.1 Opening and Closing Projects

To de-clutter the workspace and reduce the risk of accidental changes, it may be helpful to close projects that are not used being worked on. This can be done by right clicking such projects and then selecting the *Close Project* entry in the menu. Projects can similarly be re-opened using the same menu.

3.2.2 Adding Additional VDM File Extensions

It is possible to associate additional or different filename extensions with a particular VDM dialect editor, instead of the standard `.vdmsl`, `.vdmpp` and `.vdmrt`. This is done using the *Window → Preferences* menu. Click the Add button for the appropriate content type.
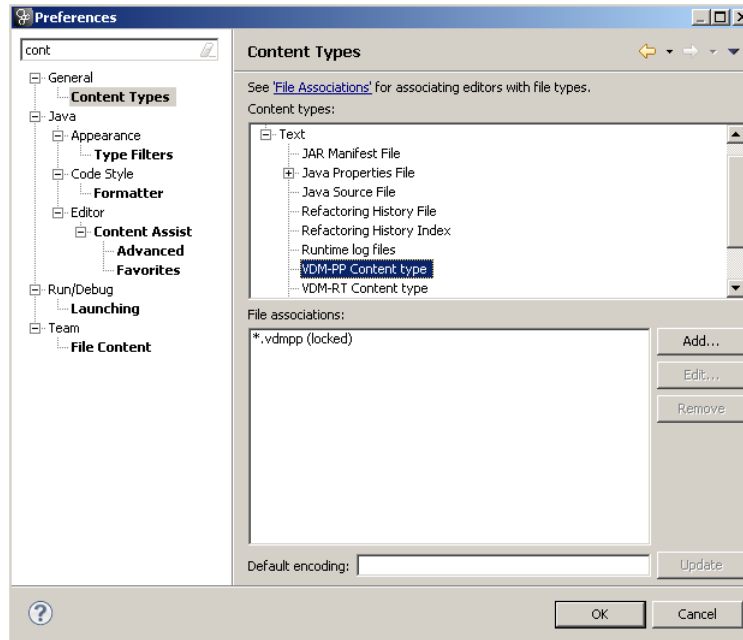
Figure 3.4: Adding Additional Contents Types

3.2.3 Filtering Project Contents

It is possible to filter out certain file types from the VDM Explorer view. This is done by clicking the small downward pointing arrow at the top right-most corner of the view. See Figure 3.5. The *Filters...* option allows various files or directories to be hidden, including directories that have no source files.

3.2.4 Including line numbers in the Editor

If line numbers are required in the dialect editors, right click in the left-hand margin of the editor and select `show line numbers` as shown in Figure 3.6. Note that the current line number and cursor position are displayed in the eclipse status bar, at the bottom of the workspace, when an editor has focus.
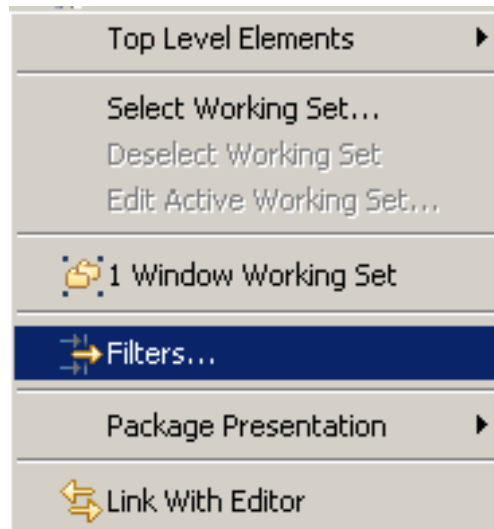
Figure 3.5: Filtering Directories without source files
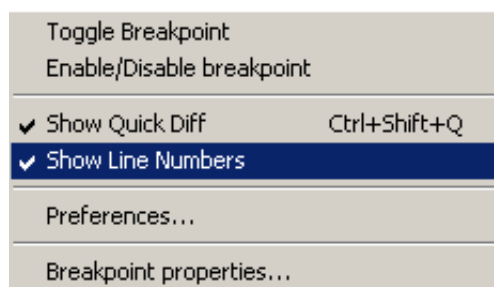
Figure 3.6: Adding Line Numbers in Editor

Chapter 4

Managing Overture Projects

4.1 Importing Overture Projects

It is possible to import Overture projects by right-clicking in the Explorer view and selecting *Import*, followed by *General → Existing Projects into Workspace*. In this way the projects from .zip files mentioned in Chapter 2 can be imported very easily.

4.2 Creating a New Overture Project

Follow these steps to create a new Overture project:

1. Create a new project by choosing *File → New → Project → Overture*;

2. Select the VDM dialect you wish to use (VDM-SL, VDM-PP or VDM-RT);

3. Click *Next*;

4. Type in a project name;

5. Choose whether you would like the contents of the new project to be in your workspace or outside (browse to the appropriate directory); and

6. Click the Finish button (see Figure 4.1).

4.3 Creating Files

Switching to the VDM perspective will change the layout of the user interface to focus on VDM development. To change perspective, go to the menu *Window → Open perspective → Other...* and choose the VDM perspective. From the this perspective you can create files using one of the following methods:

Figure 4.1: Create Project Wizard

1. Choose *File → New → VDM-SL Module* or *VDM-PP Class* or *VDM-RT Class* or

2. Right click on the Overture project where you would like to add a new file and then choose *New → VDM-SL Module* or *VDM-PP Class* or *VDM-RT Class*.

   In both cases you need to choose a file name and optionally choose a directory if you do not want to place the file in the home directory of the chosen Overture project. Then a new file with the appropriate file extension (according to the chosen dialect, `.vdmsl`, `.vdmpp` or `.vdmrt`) will be created in the directory. This file will use the appropriate module/class template to get you started. Naturally, keywords that are not required can be deleted from the template.

4.4 Adding Standard Libraries

In addition to adding new empty files it is possible to add existing standard libraries. This can be done by right-clicking on the project where the library is to be added and then selecting *New → Add VDM Library*. That will make a new window as shown in Figure 4.2. Here the different

standard libraries provide different standard functionalities. In the body of many of these functions/operations are declared as "**is not yet specified**" but the actual functionality for all of these are hard-coded into Overture so the user can get access to this when the respective standard libraries are included. This can be summarised as:

IO: This library provides functionality for input and output from/to files and the standard console.

Math: This library provides functionality for standard mathematical functions such as sine and cosine.

Util: This library provides functionality for converting different kind of VDM values mainly to and from files and strings.

CSV: This library is an extension of the IO library which provides additional functionality for saving and reading VDM values to/from comma seperate format used by excel spreadsheets.

VDM-Unit: This library provides functionality for unit testing of VDM models similar to the well-known JUnit library.
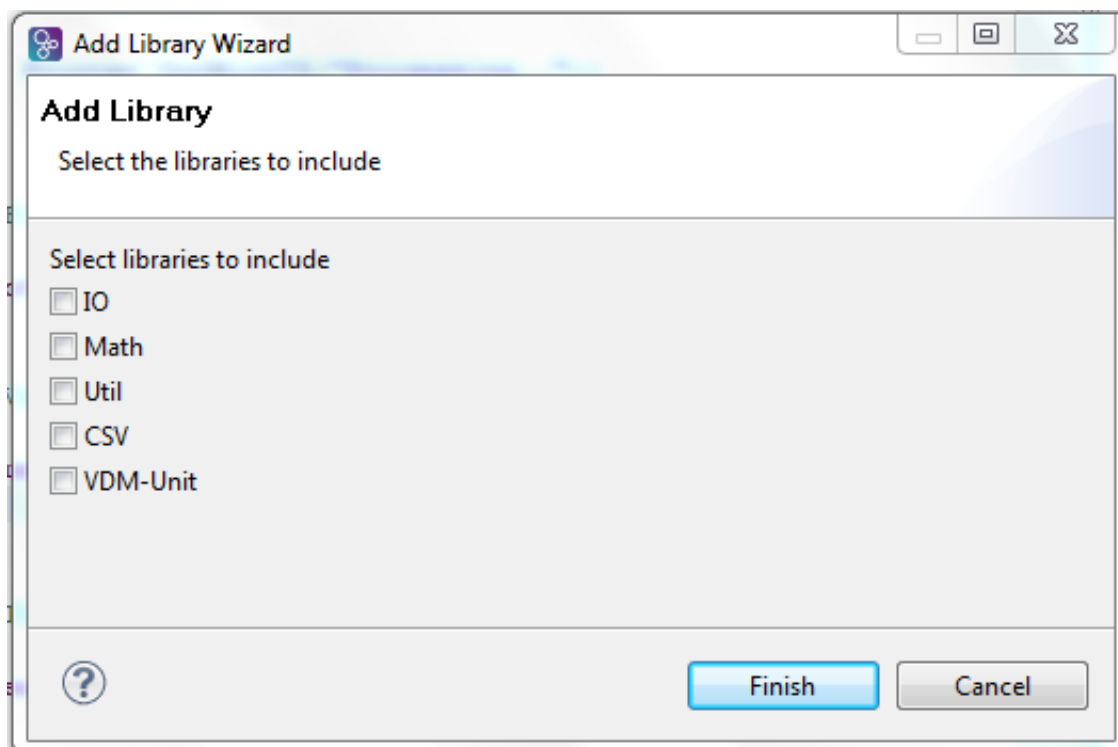

Figure 4.2: Adding New Libraries

These libraries are available for all VDM dialects also when a flat VDM-SL specification is used.

4.5   Setting Project Options

There are various VDM specific settings for an Overture project. You can change these by selecting a project in the *Explorer* view and then right clicking and selecting *Properties*. See Figure 4.3. The options that can be set for each VDM project are:
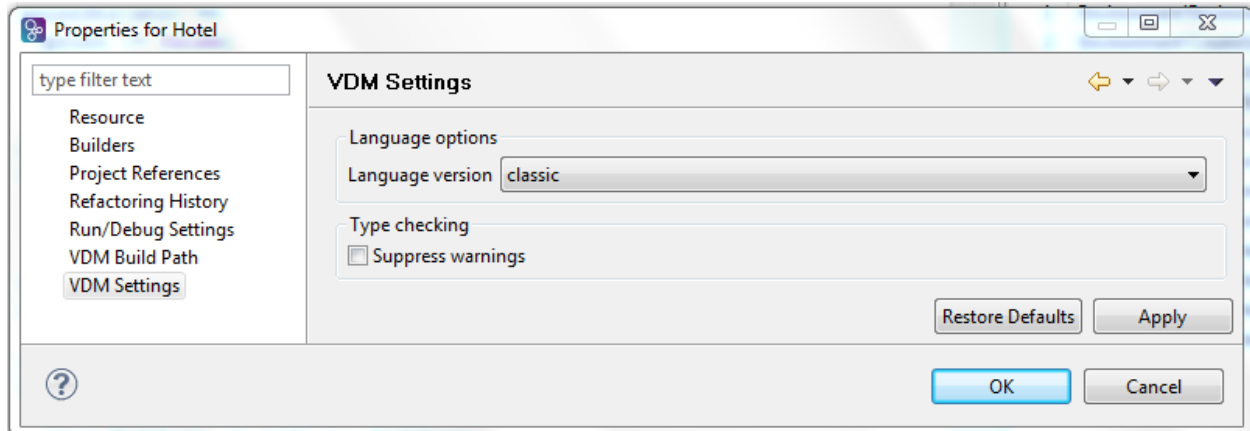
Figure 4.3: Overture Project Settings

Language version: Here the default is to use the *classic* version which is similar to that used in VDMTools. Alternatively you can select VDM-10 which is a new improved (but not necessarily backwards compatible) version of the VDM dialects developed by the Overture Language Board.

Suppress type checking warnings: Warnings are enabled by default but you can change it here.

Overture allows VDM specifications to be embedded in LaTeX files that form part of the documentation of a project as seen in Figure 4.4. The project settings allow you to define a main LaTeX file for the project, and define the order in which the different VDM source files shall be included. Note that if the "Insert coverage tables" and "Mark coverage" options are selected the LaTeX pretty printing will include test coverage information as well as provide test coverage tables for each class/module in the VDM model. It is also possible to define your own main document instead of making use of the standard one suggested by Overture (which is the name of the project followed by .tex).

It is also possible to set various preferences that apply to all projects. This is done in the general VDM preferences dialog under *Window → Preferences → VDM*. Here, for example, it is possible to link projects to VDMTools if you have the appropriate CSK executables installed on the computer. Figure 4.5 shows how it is possible to set up paths to the corresponding VDMTools executables. If these paths have been set, it is possible to right click on a project in the VDM Explorer view and select *VDM Tools → Open project in VDMTools*. Then a project file for VDM-Tools will automatically be generated with all the files from the Overture project and VDMTools
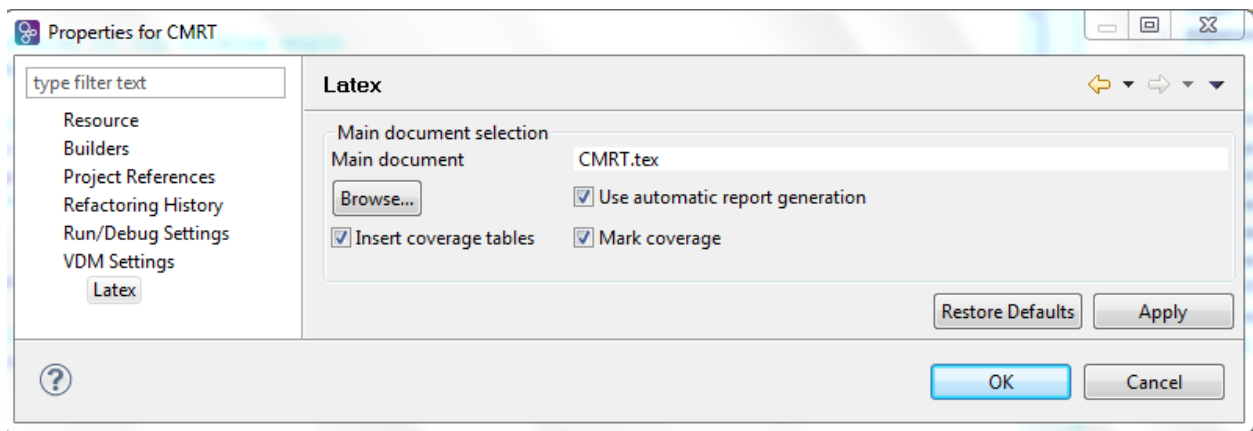
Figure 4.4: Overture Project Settings for LaTeX

will be opened. The *Preferences* dialog also allows you to switch off continuous syntax checking while editing and to set the path to *pdflatex* if this is not automatically visible from the Overture application. Finally it is possible to manage VDM templates, but that is described in Section 5.2.
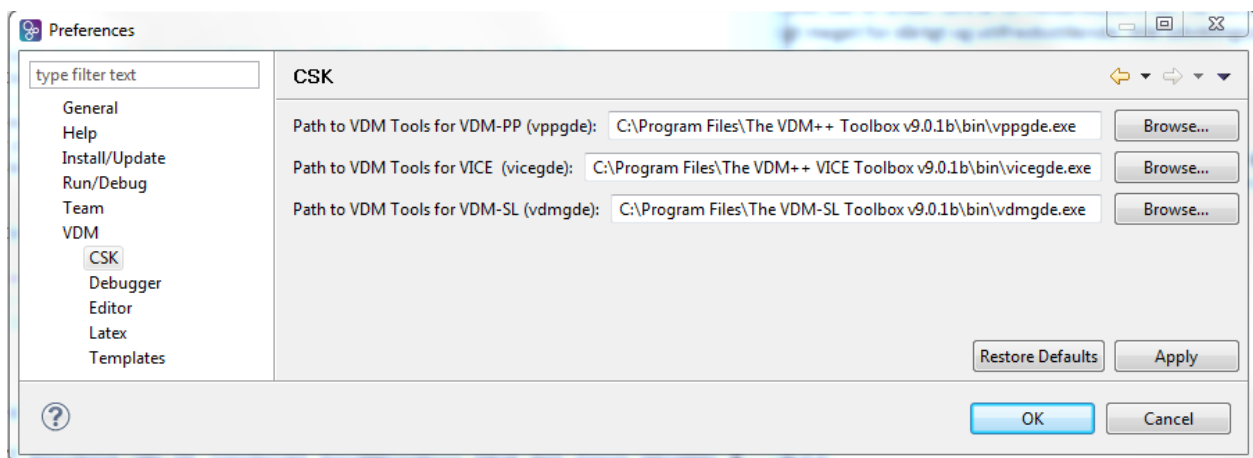
Figure 4.5: Overture Preferences for connctions to VDMTools

Chapter 5

Editing VDM Models

5.1 VDM Dialect Editors

VDM model files are always changed in the dialect Editor view. Syntax checking is carried out continuously as source files are changed (even before the files are saved). Whenever files are saved, assuming there are no syntax errors, a full type check of the *entire* VDM model is performed. Problems and warnings will be listed in the Problems view as well as being highlighted directly in the Editor view where the problems have been identified.

5.2 Using Templates

Eclipse templates can be particularly useful when writing VDM models. If you press *CTRL+space* after typing the first few characters of a template name, Overture will offer a proposal. For example, if you type "fun" followed by *CTRL+space*, the IDE will propose the use of an implicit or explicit function template as shown in Figure 5.1. The IDE includes several templates: cases, quantifications, functions (explicit/implicit), operations (explicit/implicit) and many more. The use of templates makes it much easier for users to create models, even if they are not deeply familiar with the VDM syntax.

It is possible to adjust or add to the templates defined in Overture. This can be done in the general VDM preferences under *Window → Preferences → VDM → Templates*. Figure 5.2 shows how the template for "cases" expressions is defined in Overture. Note that new templates can be added and the existing ones can be edited or removed. A full list of the standard Overture templates is available in Appendix A.

```
30  functions
31
32    NumberOfExperts: Period * Plant -> nat
33    NumberOfExperts(peri,plant) ==
34       card plant.schedule(peri)
35    pre peri in set dom plant.schedule;
36
37    ExpertIsOnDuty: Expert * Plant -> set of Period
38    ExpertIsOnDuty(ex,mk_Plant(sch,-)) ==
39      {peri| peri in set dom sch & ex in set sch(peri)};
40
41    ExpertToPage(a:Alarm,peri:Period,plant:Plant) r: Expert
42    pre peri in set dom plant.schedule and
43        a in set plant.alarms
44    post r in set plant.schedule(peri) and
45         a.quali in set r.quali;
46
47    QualificationOK: set of Expert * Qualification -> bool
48    QualificationOK(exs,reqquali) ==
49      exists ex in set exs & reqquali in set ex.quali
50
51    functionName :  parameterTypes -> resultType
52    functionName (parameterNames) == expression
53    pre preCondition
54    post postCondition
55
```

Figure 5.1: Explicit function template
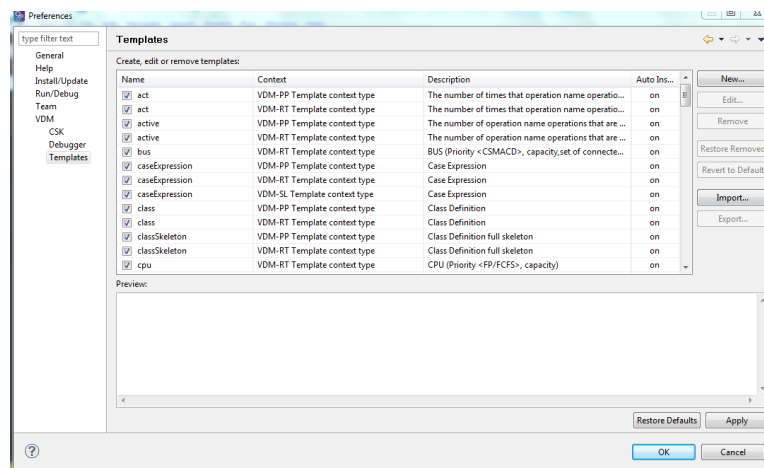
Figure 5.2: Adjusting templates for Overture

Chapter 6

Interpretation and Debugging in Overture

This section describes how to run and debug a model using the Overture IDE.

6.1 Run and Debug Launch Configurations

To execute or debug a VDM model, you must first create a launch configuration. To do this, go to the main Run menu and select *Run → Run Configurations*. Select the type of project you want to launch, click the New icon to create a new launch specification of that type and give it a name. The launch dialog requires you to identify the VDM project name, the class/module name and the initial operation/function to call in that class/module. Figure 6.1 shows a launch dialog. The standard Eclipse strategy is the launch mode called "Entry point" and then you simply click the Browse button and it will let you select a project from those available in the workspace. Clicking the Search button will search the chosen project for classes and modules to select a public operation or function from. If the chosen operation or function has parameters, the types and names of those parameters will be copied into the Operation box - these *must* be replaced with valid argument values[1].

However, there are other launch mode possibilities here as well. The "Remote Control" launch mode is advanced but it is explained in mode detail in Section 14. The "Console" launch mode enables you to get a special debug console where you can enter multiple entry points (one after another) instead of deciding upon the single entry point at launch time[2]. The commands that can be used in the "Console" view correspond to the commands you can give in VDMJ when it has been started in interpreter mode (see Section 16.3).

Your new launch configuration can be started immediately by clicking the *Run* button at the bottom of the dialog. Alternatively, the configuration can simply be saved by clicking *Apply*. Once a launch configuration has been defined, it can be re-run at any time by using the small downward arrow next to the run or debug icons (   ) in the IDE toolbar.

---

[1]You will see type checking errors at the top of the dialog if you do not do this, such as "Error 3063: Too few arguments in ..."

[2]Those familiar with VDMTools will recognise this functionaility as initialising a specific VDM model and then having a prompt where different expressions can be evaluated making use of the definitions from the model.

A launch configuration can either be started normally, which will simply evaluate the expression given and stop, or it can be started in debug mode, which will stop the evaluation at any breakpoints you may have set. The same launch configuration can be used for either purpose, though by default those created through the *Run Configurations* dialog will appear in the favourites list under the *Run* toolbar icon. Similarly, a launch configuration created under the *Debug Configurations* dialog will appear under the favourites of the debug toolbar icon. You can control which icons display the launch configuration in the *Common* tab on the dialog. This is standard Eclipse behaviour.
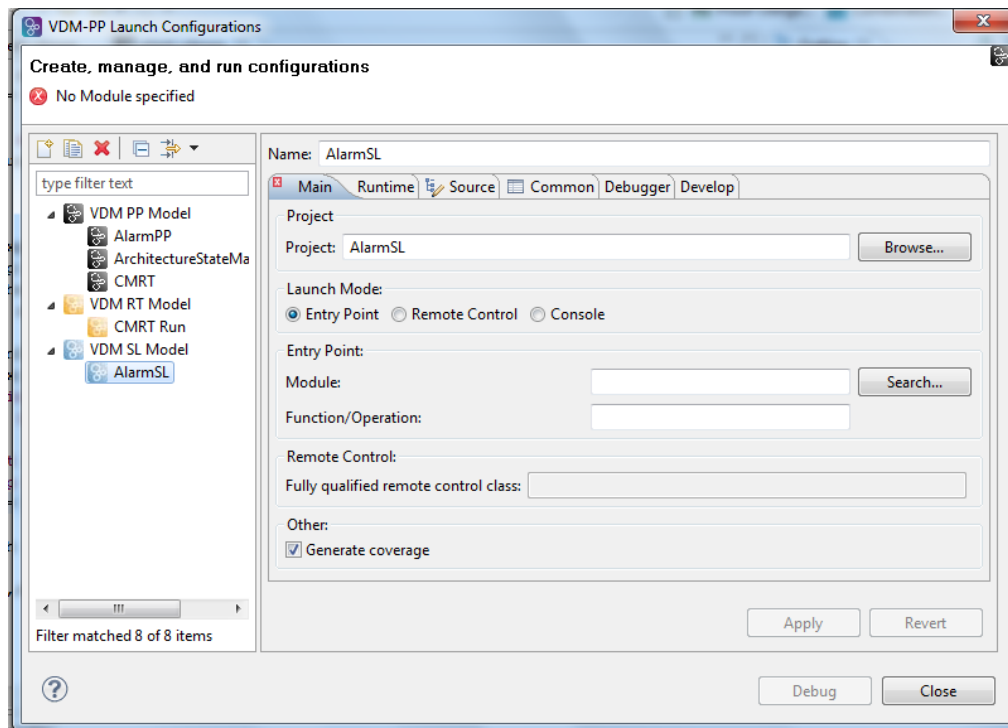
Figure 6.1: The launch configuration dialog

Whenever a launch configuration is started up it is also possible to decide upon which additional run-time checks to carry out. Per default all possible run-time checks are swiched on but if desired (some of) these can be swiched off using the "Runtime" pane (see Figure 6.2). Note that for VDM-RT debugging it is also possible to switch off the logging of all events appearing during the debugging. The different run-time checks that can be performed are:

Dynamic type checks: This is an option for the interpreter (default on) to continuously type check values during interpretation of a VDM model. It is possible to switch off the check here.

Invariant checks: This is an option for the interpreter (default on) to continuously check both state and type invariants. It is possible to switch off this check here, but note that option requires dynamic type checking also to be switched off.

Pre condition checks: This is an option for the interpreter (default on) to continuously check pre-conditions for all functions and operations during interpretation of a VDM model. It is possible to switch off this check here.

Post condition checks: This is an option for the interpreter (default on) to continuously check post-conditions for all functions and operations during interpretation of a VDM model. It is possible to switch off this check here.

Measure checks: This is an option for the interpreter (default on) to continuously check recursive functions, for which a measure function has been defined. It is possible to switch off this check here.

In the launch configuration the "Debug" pane shown in Figure 6.3 can also be useful in rare cases where one have particular deep recursion for example. this is an advanced setting where one can decide the arguments given to the Java virtual machine for allocation of maximum amounts of space per thread in a VDM model. However, this option is rarely used.
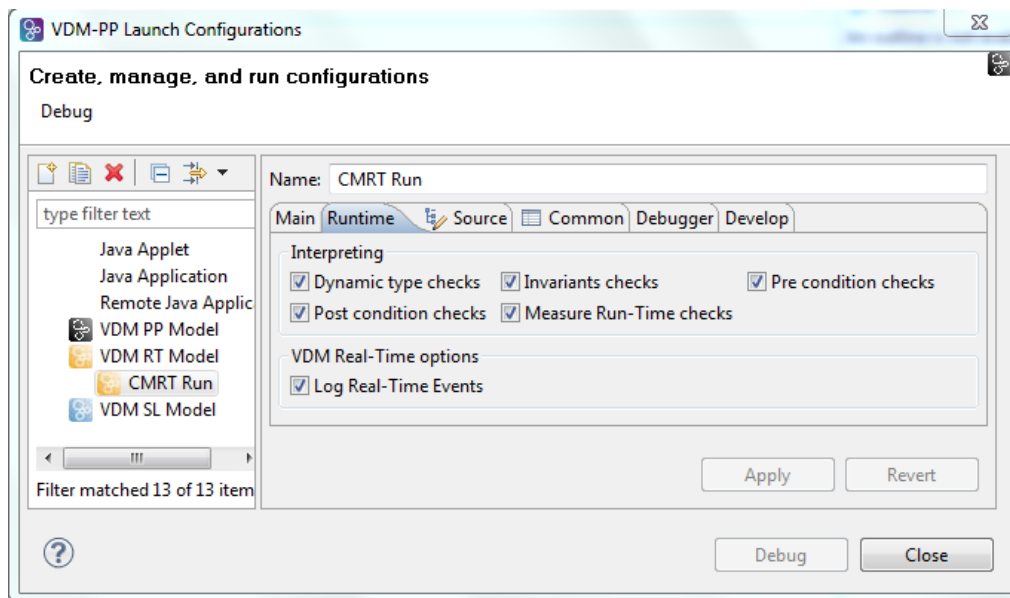
Figure 6.2: The launch configuration dialog

6.2 The Debug Perspective

The Debug Perspective contains all the views commonly needed for debugging in VDM. Breakpoints can easily be set in the model by double clicking in the left margin of the Editor view at the chosen line. When the debugger reaches the location of a breakpoint and stops, you can inspect the values of different identifiers and step through the VDM model line by line.
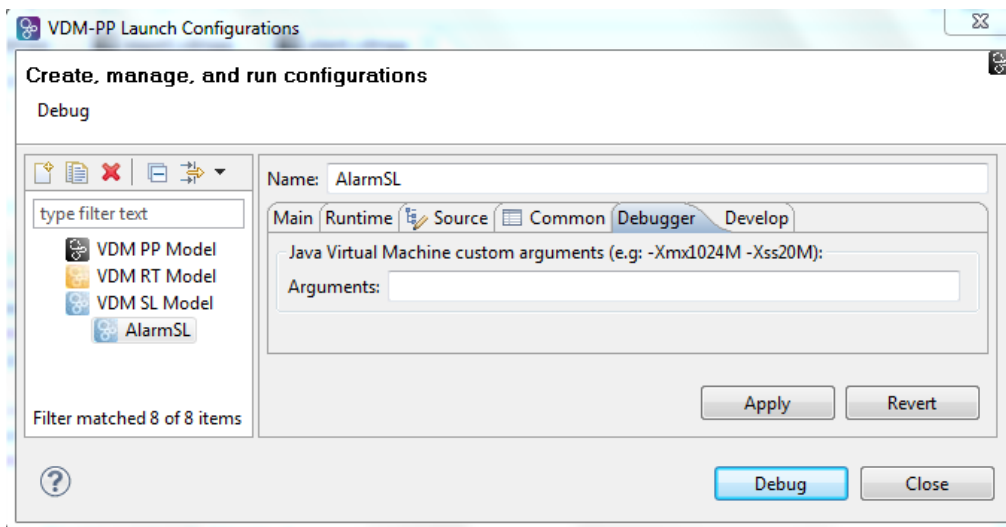
Figure 6.3: The launch configuration dialog

The Debug Perspective is illustrated in Figure 6.4
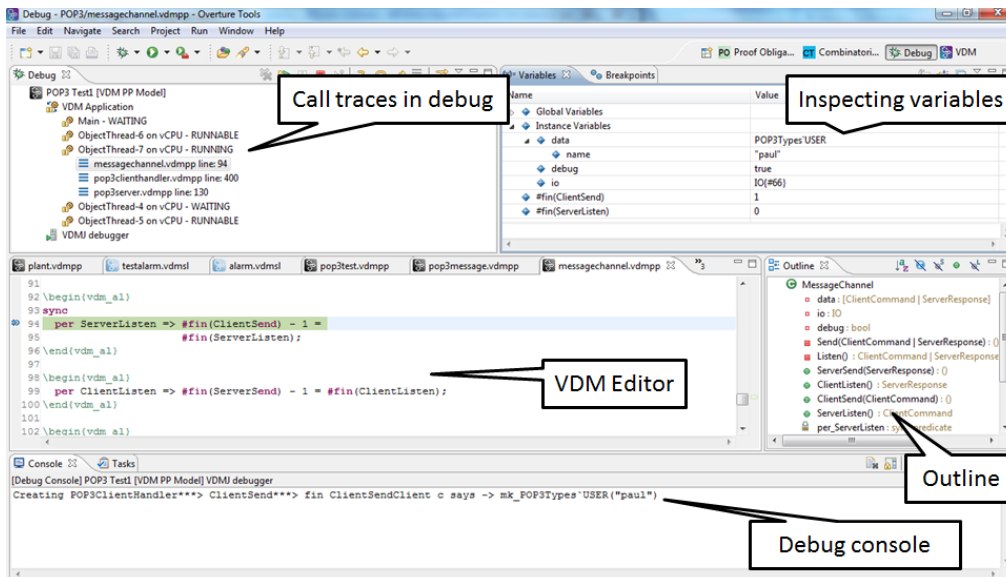
Figure 6.4: Debugging perspective

Table 6.1: Overture debugging buttons

Button	Explanation
	Resume debugging
	Suspend debugging
	Terminate debugging
	Step into
	Step over
	Step return
	Use step filters

6.2.1 The Debug View

The *Debug* view is located in the upper left corner in the Debug perspective – see Figure 6.4. The view shows all running models and whether a given model is stopped, suspended or running. It shows the call stack of models that are suspended, and for VDM++ and VDM-RT stacks for all threads are shown. At the top of the view, there are buttons for debugging such as: stop, step into, step over, resume, etc. (see Table 6.1). Note that in case a multi-threaded VDM model is debugged it is possible in this view to change to another thread to inspect where it is currently and inspect the local variables at that thread since they are all stopped when a breakpoint is reached.

6.2.2 The Variables View

The *Variables* view shows all the variables in a thread context, allowing them to be examined after a breakpoint (or an error) has been reached. The variables and their values are automatically updated when stepping through a model. The view is located in the upper right hand corner in the Debug perspective. It is possible to inspect compound variables, expand nested structures and so on. Note that when you stop at a permission predicate it is also possible to see the value of the relevant history counters (in Figure 6.4 `#fin(ClientSend)` and `#fin(ServerListen)`). By right-clicking on a variable it is possible to select a "watch point". As a result a window like Figure 6.5 will occur. Using this it is possible to watch the value of such a variable easily whenever a new stop is reached in the debugging process.

6.2.3 The Breakpoints View

Breakpoints can be added in any perspective from the Editor view[3]. The debug perspective also has a *Breakpoints* view that lists all current breakpoints, allowing you to navigate easily to the location

---

[3]Note that breakpoints can only be set on lines that contain executable code.

25

Name	Value
X+Y =? "b2"	map[4]
◆ Maplet 1	{<D> |-> 4}
◆ Maplet 2	{<E> |-> 1}
◆ Maplet 3	{<A> |-> 1}
◆ Maplet 4	{<C> |-> 5}
✚ Add new expression	

Figure 6.5: Example of a watchpoint

of a given breakpoint, disable it or delete it. The view is located in the same panel as the Variables view in the upper right hand corner.

6.2.4 Conditional Breakpoints

Breakpoints can be conditional. This is a powerful feature for the developer since it allows you to specify a conditional expression which has to be true for the debugger to stop at the given breakpoint. As well as using an expression, a conditional breakpoint may specify a hit count and whether the breakpoint should stop when the hit count is equal to, greater than, or a multiple of the given value, or a general expression using the variables in scope at the breakpoint.

A normal breakpoint can be made conditional by right clicking on the breakpoint mark in the Editor view[4] and selecting *Breakpoint Properties*. This opens a dialog like the one shown in Figure 6.6.

6.2.5 The Expressions View

The *Expressions* view allows you to define expressions that are evaluated whenever the debugger stops. Watched expressions can be added to the view directly, or created by selecting *Watch* when right-clicking a variable in the Variables view. It is also possible to edit existing expressions. The view sits in the same panel as the Breakpoints view and the Variables view.

---

[4]Note this is not possible from the Breakpoint view
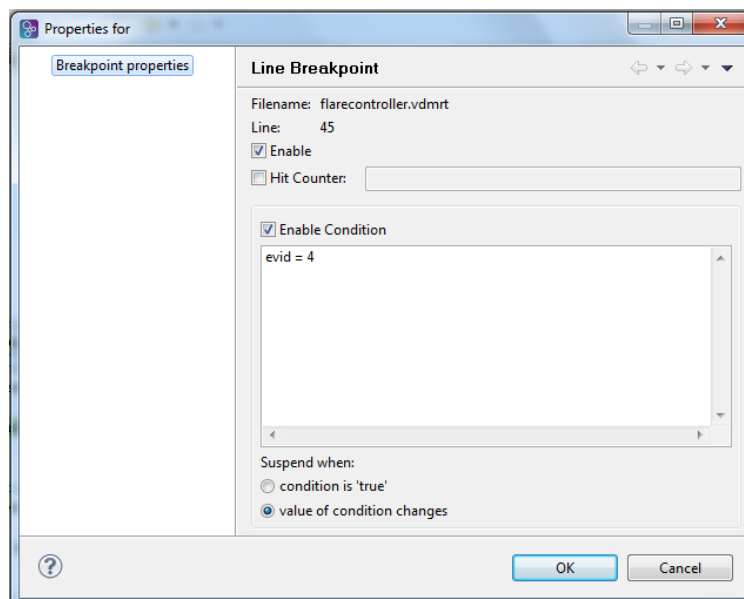
Figure 6.6: Conditional breakpoint options

Chapter 7

Collecting Test Coverage Information

When a VDM model is being interpreted, it is possible to automatically collect test coverage information. Test coverage measurements help you to see how well a given test suite exercises your VDM model.

In order to enable the collection of test coverage data, go to the debug launch configuration and select the *Generate coverage* option. After running this configuration, a new file with a `.cov` extension will be created for each file in the project. These files are written into a project subfolder named `generated/coverage/<date and time>`. Double-clicking the `.cov` files will open a special editor window that displays the source with coverage coloured in red/green (red is executable but not covered). Alternatively, a PDF file containing the entire model with coloured test coverage summarised for all runs can be generated by right-clicking on the project name and selecting *Latex → Latex Coverage*.

Chapter 8

Pretty Printing to LaTeX

It is possible to use literate programming/specification [Johnson96] with Overture just as you can with VDMTools. To take advantage of this, you need to use the LaTeX text processing system with plain VDM models mixed with textual documentation. The VDM model parts must be enclosed within "`\begin{vdm_al}`" and "`\end{vdm_al}`". The text-parts outside these specification blocks are ignored by the VDM parser, though note that each source file must start with a recognizable LaTeX construct: a `\documentclass, \section, \subsection` or a LaTeX comment.

Chapter 9

Managing Proof Obligations

In all VDM dialects, Overture can identify places where run-time errors *could* potentially occur if the model was to be executed. The analysis of these areas can be considered as a complement to the static type checking that is performed automatically. Type checking accepts specifications that are *possibly* correct, but we also want to know the places where the specification could possibly fail.

Unfortunately, it is not always possible to statically check if such potential problems will *actually* occur at run-time error or not. So Overture creates *Proof Obligations* for all the places where run-time errors *could* occur. Each proof obligation (PO) is formulated as a predicate that must hold at a particular place in the VDM model if it is error-free, and so it may have particular context information associated with it. POs can be considered as constraints that will guarantee the internal integrity of a VDM model if they are all met. In the long term, it will be possible to prove these constraints with a proof component in Overture, but this is not yet available.

POs can be divided into different categories depending upon their nature. The full list of categories can be found in Appendix G along with a short description for each of them.

The proof obligation generator is invoked either on a VDM project (and then POs for all the VDM model files will be generated) or for one selected VDM file. Right-click the project or file in the Explorer view and then select *Proof Obligations → Generate Proof Obligations*. Overture will change into a special *Proof Obligations* perspective as shown in Figure 9.1.

Note that in the *Proof Obligation Explorer* view, each proof obligation has four components:

- A unique number in the list shown;

- The name of the definition in which the proof obligation is located;

- The proof obligation category (type); and

- A status field indicating whether the proof obligation is trivially correct or would have to be proved by a proof engine.

At the top of the *Proof Obligation Explorer* the *Filter proved* button allows you to filter away all the proof obligations that are trivially correct.

Figure 9.1: The Proof Obligation perspective

Chapter 10

Combinatorial Testing

In order to better automate the testing process, a notion of test *traces* has been introduced into VDM++[1]. Traces are effectively regular expressions that can be expanded to a collection of test cases. Each test case comprises a sequence of operation calls. If a user defines a trace it is possible to make use of a special *Combinatorial Testing* perspective to automatically expand the trace and execute all of the resulting test cases. Subsequently, the results from the tests can be inspected and erroneous test cases easily found. You can then fix problems and re-run the trace to check they are fixed.

10.1   Using the Combinatorial Testing GUI

The syntax for trace definitions is defined in the VDM-10 Language Manual. If you have created a `traces` entry for a module or class it can be executed via the *Combinatorial Testing* perspective. See Figure 10.1.

Different icons are used to indicate the verdict in a test case. These are:

🔍: This icon is used to indicate that the test case has not yet been executed.

✔: This icon is used to indicate that the test case has a pass verdict.

ⓘ: This icon is used to indicate that the test case has an inconclusive verdict.

✖: This icon is used to indicate that the test case has a fail verdict.

▷ S4 (2800 skipped 120): If any test cases result in a run-time error, other test cases with the same sequence of calls will be filtered and automatically skipped in the test execution. The number of skipped test cases is indicated after the number of test cases for the trace definition name.

In the CT Overview view, you can right-click on any individual test case and then execute it with the interpreter (see Figure 10.2). This is particularly useful for failed test cases since the interpreter allows you to step through the evaluation to the place where it is failing. You can inspect the exact circumstances of the failure, including the values of the different variables in scope.

---

[1]Note that this is only available for VDM-SL models if the VDM-10 language version has been selected
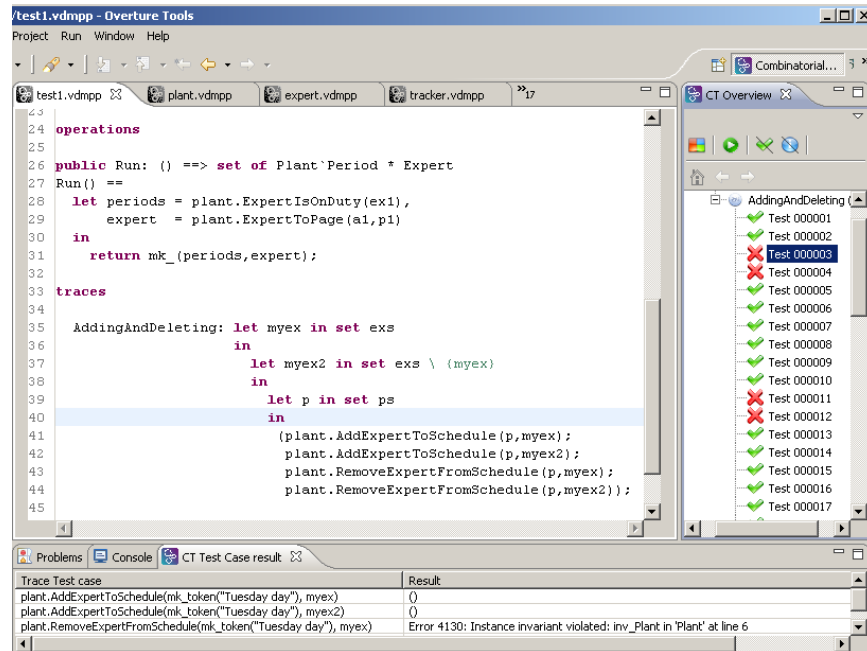
Figure 10.1: Using Combinatorial Testing
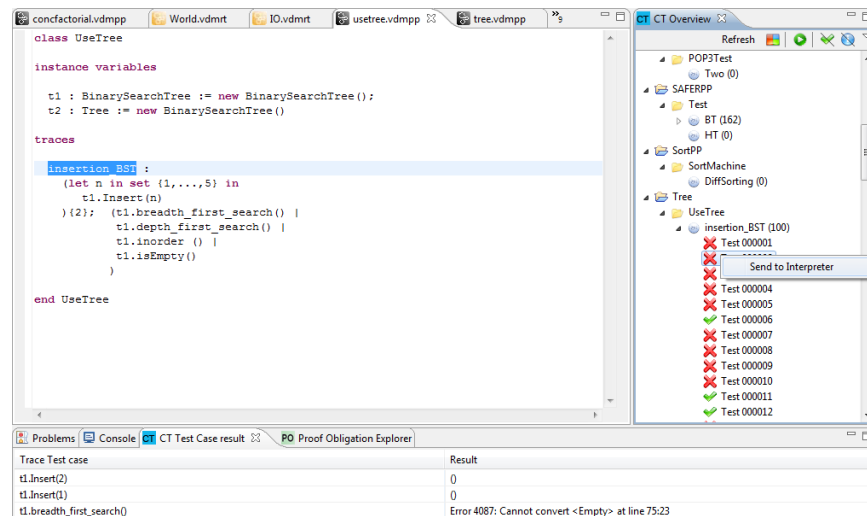
Figure 10.2: Moving test case from Combinatorial Testing to Interpreter

Chapter 11

Mapping VDM++ To and From UML

VDM++ and VDM-RT projects can be converted automatically back and forth between VDM and the corresponding UML model[1]. Essentially, VDM and UML can be considered as different views of the same model. A UML model is typically used to give a graphical overview of the model using class diagrams, and sequence diagrams can be used to indicate the test scenarios that a user would like to perform. The VDM model is typically used to define the implementation and constraints for each class and is therefore used for detailed semantic analysis.

The exchange between VDM and UML is done using the XML format called XMI. At the moment, Enterprise Architect is the only UML tool supported. Export from EA is done by selecting the *Project* menu and *Import/Export → Export Package to XMI*. This is illustrated in Figure 11.1.

Importing and exporting a UML model is an option in the Overture *Explorer* view, where right-clicking a VDM++ or VDM-RT project gives a submenu for *UML Transformation*. From here it is possible to *Import XMI* or to *Export XMI*.

---

[1]In the current version of Overture this feature is somewhat unstable.
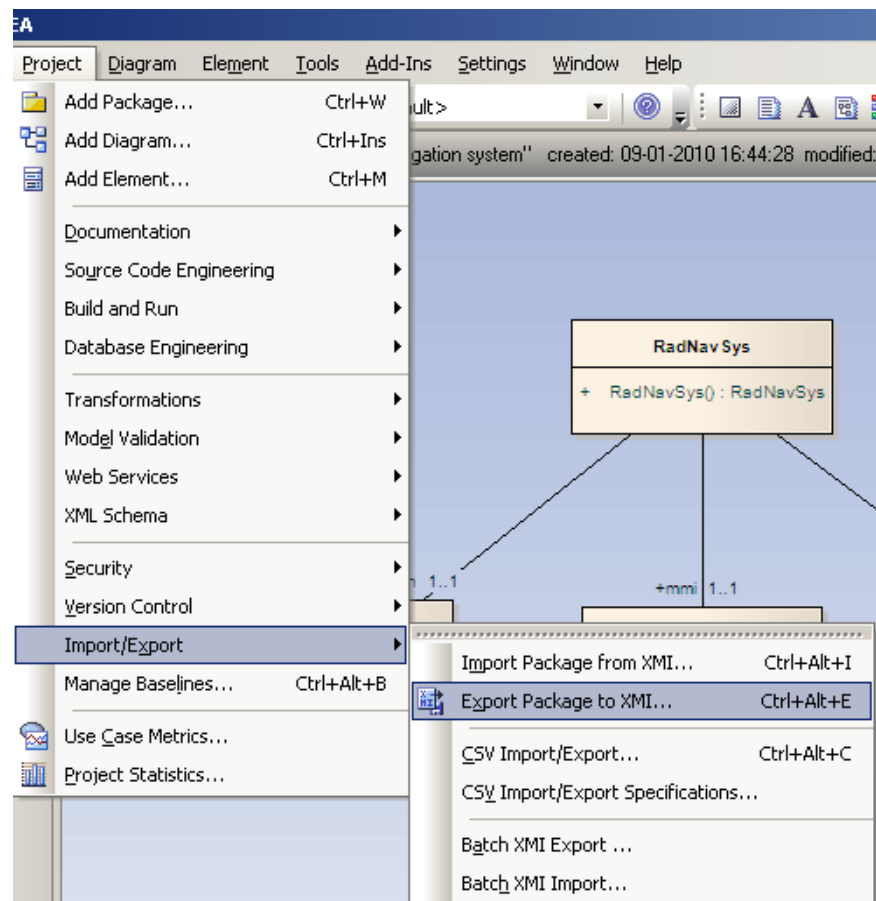
Figure 11.1: Exporting UML definitions from EA

Chapter 12

Moving from VDM++ to VDM-RT

The methodology for the development of distributed real-time embedded systems in VDM defines a step where you move from an initial VDM++ model to a VDM-RT model [Larsen&09]. This step is supported by the Overture tool which will convert a VDM++ project to create the starting point for a new VDM-RT project. This is done by right-clicking on the VDM++ project to be converted in the Explorer view, followed by the *Clone as VDM-RT* option. A new VDM-RT project is then automatically created. It will have the same name as the original VDM++ project, but with VDM_RT appended. Inside the project, all the .vdmpp files will have been converted to a .vdmrt extension. The original VDM++ project is not changed at all. So this is simply a quick and easy way to get to the starting point for a VDM-RT model. You would then manually create a system class with appropriate declarations of CPUs and BUSses and proceed with the real time model development.

Chapter 13

Analysing and Displaying Logs from VDM-RT Executions

Whenever a VDM-RT model is executed, a logfile is created in a `generated/logs/<launch>` subfolder with a `.logrt` extension. The file name for the logfile indicates the time at which the model was executed, so it is possible to distinguish multiple runs. Logfiles can be viewed with the built-in *RealTime Log Viewer*, by double-clicking the .logrt file in the Explorer view. The log viewer enables you to explore the simulated system execution in various ways. In Figure 13.1 the architectural overview of the system is shown, describing the CPU and BUS topology of the model.
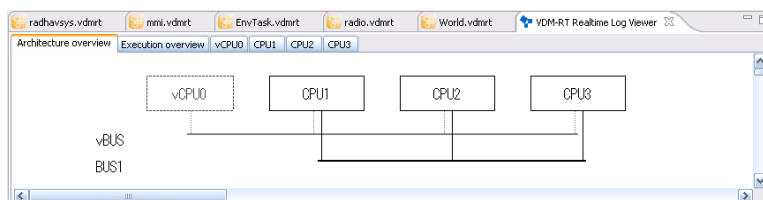
Figure 13.1: Architectural overview

The log viewer also enables you to get an overview of the model execution at a system level – see Figure 13.2. This view shows how the different CPUs communicate via the BUSes of the system.

Since the complete execution of a model cannot generally be shown in a normal sized window, you have the option of jumping to a certain time index using the *Go to time* button. It is also possible to export all the generated views to JPEG format files using the *Export Image* button. All the generated images will be placed in the same folder as the `.logrt` file.

The log viewer can also give an overview of all executions on a single CPU. This view gives a detailed description of all operations and functions invoked on the one CPU as well as the scheduling of concurrent processes. See Figure 13.3.
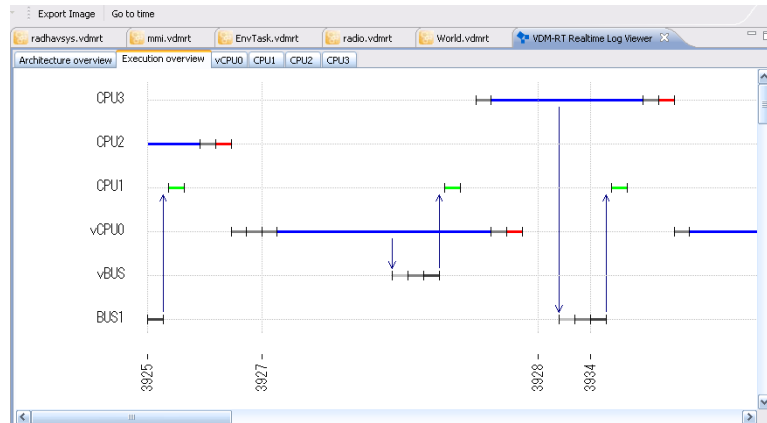
Figure 13.2: Execution overview
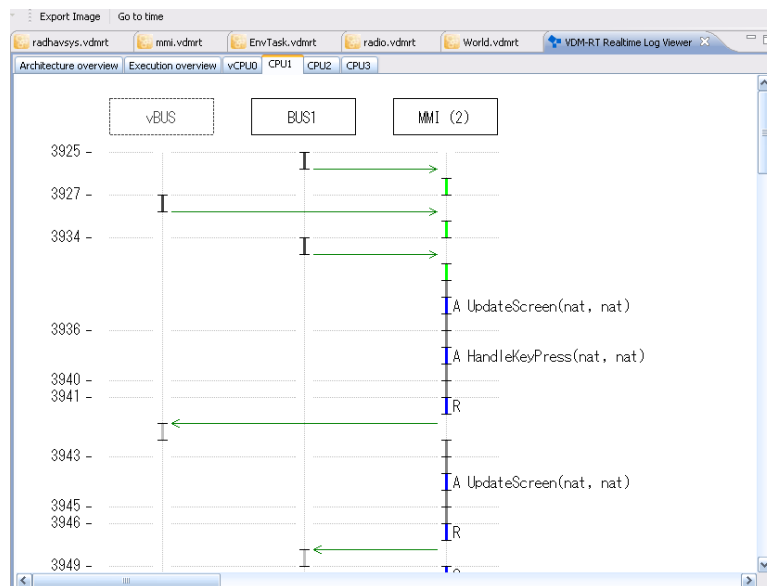
Figure 13.3: Execution on single CPU

Chapter 14

Defining Your Own Java Libraries to be used from Overture

VDM models are not appropriate for describing everything. It is common to have existing legacy code that you may not wish to spend time modelling in VDM, but would like to make use of from a VDM model. Overture has a feature to link a VDM model with external Java libraries contained in a standard `jar` file [1]. Using this feature it is possible to call functionality provided by jar files from a VDM model. This functionality corresponds to DL modules/classes in VDMTools [DLMan].

External jar libraries are linked to VDM via `is not yet specified` statements and expressions. Operations or functions of modules or classes can be delegated to an external jar, calling out to a Java class. The Java delegate, if present, has the same name as the VDM module/class name with underscores ("_") replaced with package naming dots ("."). For example, the VDM class `remote_lib_sensor` becomes the class `remote.lib.sensor` in Java. The delegate lookup is only done once and only when an `is not yet specified` statement or expression is first reached in a class or module. The jar with the external library must be placed in the VDM project in a subfolder named `lib` where it will be put in the class-path of the interpreter when it is executed.

14.1 External Library Example

In this example, a remote sensor will be defined in VDM which can read a value from a real sensor. The VDM model interface of the sensor can be seen in listing 14.1 and the Java class implementing it can be seen in listing 14.2. The values that are to be exchanged between the Overture IDE and the jar file needs to be the internal *Value* objects used in VDMJ. Documentation about these classes can be found in the VDMJ Design Specification [Battle10].

```
class remote_lib_sensor

operations
```

---

[1] In fact the `IO`, `MATH`, `Util`, `CSV` and `VDM-Unit` libraries are implemented as such external jar files.

```
public getValue : int ==> int
getValue (id) == is not yet specified;


end remote_lib_sensor
```

Listing 14.1: Remote sensor VDM class

```
package remote.lib;

import org.overturetool.vdmj.runtime.ValueException;
import org.overturetool.vdmj.values.IntegerValue;
import org.overturetool.vdmj.values.Value;

public class sensor
{
        public Value getValue(Value id) throws ValueException
        {
                int result = ... // Read a value for sensor number "id"
                return new IntegerValue(result);
        }
}
```

Listing 14.2: Remote sensor Java class

Chapter 15

Enabling Remote Control of the Overture Interpreter

In some situations, it may be valuable to be able to establish a front end (for example a GUI or a test harness) for calling a VDM model. This feature corresponds roughly to the CORBA based API from VDMTools [APIMan].

A VDM model can be remotely controlled by implementing the Java interface `RemoteControl`. Remote control should be understood as a delegation of control of the interpreter, which means that the remote controller is in charge of the execution or debug session and is responsible for taking action and executing parts of the VDM model when needed. When finished, it should return and the session will stop. When a Remote controller is used, the Overture debugger continues working normally, so for example breakpoints can be used in debug mode. A debugging session with the use of a remote controller can be started by placing the jar with the RemoteControl implementation in a project subfolder called `lib`. The fully qualified name of the RemoteControl class must then be specified in the launch configuration in the *Remote Control* box.

15.1   Example of a Remote Control Class

In this example, we have a VDM class `A` which defines an operation that just returns its argument. As seen in listing 15.1, it is possible to call `execute` on the Overture interpreter via the RemoteInterpreter object which is passed to the RemoteControl implementation via the `run` method. The method returns a string with the result. A more advanced `valueExecute` method is also available which returns the internal Value type of the interpreter which is useful for more complex results. The values exchanged between the Overture IDE and the controller are the internal Values used in VDMJ. Documentation about these can be found in the VDMJ Design Specification [Battle10].

```
import org.overturetool.vdmj.debug.RemoteControl;
import org.overturetool.vdmj.debug.RemoteInterpreter;

public class RemoteController implements RemoteControl
```

```
{
        public void run(RemoteInterpreter interpreter) throws Exception
        {
                System.out.println("Remote controller run");
                System.out.println("The answer is " +
                        interpreter.execute("1 + 1"));
                System.out.println("The answer is " +
                        interpreter.execute("new A().op(123)"));
                System.out.println("The answer is " +
                        interpreter.execute("new A().op(1 + 3)"));
        }
}
```

Listing 15.1: Remote Controller Java class

Chapter 16

A Command-Line Interface to VDMJ

At the centre of the Overture tool there is a Java implementation of VDM called *VDMJ*. This provides a command-line interface that may be valuable as it can be used independently of the Eclipse interface of Overture.

16.1 Starting VDMJ

VDMJ is contained entirely within one jar file. The jar file contains a MANIFEST that identifies the main class to start the tool, so the minimum command line invocation is as follows:

```
$ java -jar vdmj-2.0.2.jar
VDMJ: You must specify either -vdmsl, -vdmpp or -vdmrt
Usage: VDMJ <-vdmsl | -vdmpp | -vdmrt> [<options>] [<files>]
```

The first parameter indicates the VDM dialect to use and then various extra options can be used. These are:

-r: This indicates the VDM release number (classic or vdm10).

-w: This will suppress all warning messages.

-q: This will suppress all information messages, such as the number of source files processed etc.

-i: This will start the command line interpreter if the VDM model is successfully parsed and type checked, otherwise the errors discovered will be listed.

-p: This will generate all proof obligations for the VDM model (if it is syntax and type correct) and then stop.

-e <exp>: This will evaluate the <exp>, print the result, and stop.

-c <charset>: This will select a file character set, to allow a specification written in languages other than the default for your system.

-t <charset>: This will select a console character set. The output terminal can use a different character set to the specification files.

-o <filename>: This will save the internal representation of a parsed and type checked specification. Such files are effectively libraries, and can be can be re-loaded without the parsing/checking overhead. If files are sufficiently large, this may be faster.

-pre: This will disable all pre-condition checks.

-post: This will disable all post-condition checks.

-inv: This will disable type/state invariant checks.

-dtc: This will disable all dynamic type checking.

-measures: This will disable recursive measure checking.

-log: This will enable VDM-RT real-time event logging (see Chapter 13).

-remote: This enables remote control of the VDMJ executable.

Normally, a VDM model will be loaded by identifying all of the VDM source files to include. At least one source file must be specified unless the -i option is used, in which case the interpreter can be started with no specification. If a directory is specified rather than a file, then VDMJ will load all files in that directory with a suffix that matches the dialect (eg. `*.vdmpp` files for VDM++). Multiple files and directory arguments can be mixed.

If no `-i` option is given, the tool will only parse and type check the VDM model files, giving any errors and warnings on standard output, then stop.

The `-p` option will run the proof obligation generator and then stop, assuming the specification has no type checking errors.

For batch execution, the `-e` option can be used to identify a single expression to evaluate in the context of the loaded specification, assuming the specification has no type checking errors.

16.2   Parsing, Type Checking, and Proof Obligations in VDMJ

All specification files loaded by VDMJ are parsed and type checked automatically. There are no type checking options; the type checker always uses `possible` semantics. If a specification does not parse and type check cleanly, the interpreter cannot be started and proof obligations cannot be generated (though warnings are allowed). All warnings and error messages are printed on standard output, even with the `-q` option.

A source file may contain VDM definitions embedded in a LATEX file using `vdm_al` environments (see Chapter 8); the markup is ignored by the parser, though reported line numbers will be correct. Note that each source file must start with a recognizable LATEX construct: a `\documentclass`, `\section`, `\subsection` or a LATEX comment.

The VDMJ Java process will return with an exit code of zero if the specification is clean (ignoring warnings). Parser or type checking errors result in an exit code of 1. The interpreter and PO generator always exit with a code of zero.

16.3 The VDMJ Interpreter and Debugger

Assuming a specification does not contain any parse or type checking errors, the interpreter can be started by using the `-i` command line option. The interpreter is an interactive command line tool that allows expressions to be evaluated in the context of the specification loaded. For example, to load and interpret a VDM-SL specification from a single file called `shmem.vdmsl`, the following options would be used:

```
$ java -jar vdmj-2.0.2.jar -vdmsl -i shmem.vdmsl
Parsed 1 module in 0.266 secs. No syntax errors
Type checked in 0.047 secs. No type errors
Interpreter started
```

The interpreter prompt is ">". The interactive interpreter commands are as follows (abbreviated forms are permitted for some, shown in square brackets):

modules: This command lists the loaded module names in a VDM-SL specification. For a flat VDM-SL model, the single name DEFAULT is used. The default module will be indicated in the list displayed.

classes: This command lists the loaded class names in VDM++ and VDM-RT specifications. The default class will be indicated in the list displayed.

default <module/class>: This command sets the default module/class name as the prime scope for which the lookup of identifiers appear (i.e. names in the default module do not need to be qualified, so you can say "print xyz" rather than "print M'xyz").

create <id> := <exp>: This command is only available for the VDM++ and VDM-RT dialects. It creates a global variable that can be used subsequently in the interpreter. It is mostly used for creating global instances of classes.

log [<file> | off]: This command can only be used with VDM-RT models. It starts to log real-time events to the file indicated. By default, event logging is turned off. Logging can be directed to the console by using log with no arguments, or to a file using `log <filename>`. Logging can subsequently be turned off again by using `log off`. The events logged include requests, activations and completions of all functions and operations, as well as all object creations, creation of CPUs and BUSses, deployment of objects to specific CPUs and the swapping in/out of threads.

state: This command can only be used for the VDM-SL dialect and shows the default module state. The value of the state can be changed by operations called.

[p]rint <expression>: This command evaluates the expression provided in the current context.

runtrace <name> [test number]: This command runs the trace called <name>. This will expand the combinatorial test and execute the resulting operation sequences. If a specific test number is provided, only that one test from the expansion will be executed.

debugtrace <name> [test number]: This command is the same as runtrace, except that if a runtime exception is encountered during the execution of a test, control will enter the debugger. With runtrace, runtime exceptions are recorded as the result of a (failed) test, rather than trapping into the debugger.

filter %age | <reduction type>: This command reduces the size of expanded CT traces to a given percentage (eg. 10%). There are various options for making the actual selection of tests to remove: "RANDOM", "SHAPES_NOVARS", "SHAPES_VARNAMES" or "SHAPES_VARVALUES" (the names are not case sensitive).

assert <file>: This command runs assertions from the file provided. The assertions in the file must be Boolean expressions, one per line. The command evaluates every assertion in the file, raising an error for any which is false.

init: This command re-initializes the global environment. Thus all state components will be initialised to their initial value again, created variables are lost and code coverage information is reset.

env: This command lists the value of all global symbols in the default environment. This will show the signatures for all functions and operations as well as the values assigned to identifiers from value definitions and global state definitions (in VDM++ terminology, public static instance variables). Note that this includes invariant, initialization and pre/postcondition functions. In the VDM++ and VDM-RT dialects, the identifiers created using the create command will also be included.

pog [<fn/op>]: This command generates a list of all proof obligations for the VDM model that is loaded. There is an optional argument to indicate one function or operation name.

break [<file>:]<line#> [<condition>]: This command creates a breakpoint at a specific file and line and optionally makes it a conditional breakpoint.

break <function/operation> [<condition>]: This command creates a breakpoint at the start of the body of a named function or operation and optionally makes it a conditional breakpoint.

trace [<file>:]<line#> [<exp>]: This command creates a tracepoint for a specific file and line. A tracepoint prints the value of the expression given whenever the tracepoint is reached, and then continues.

trace <function/operation> [<exp>]: This command create a tracepoint at the start of a function or operation body. See `trace` above for an explanation of tracepoints.

remove <breakpoint#>: This command removes a trace/breakpoint by referring to its number (given by the `list` command).

list: This command provides a list of all current trace/breakpoints by number.

coverage [clear|write <dir>|merge <dir>|<filenames>]: This command manages test coverage information. The coverage command displays the source code of the loaded VDM model (by default, all source files are listed), with "+" and "-" signs in the left hand column indicating lines which have been executed or not. The percentage coverage of each source file is displayed. Typically, the testing of a specification will be incremental, and so it is convenient to be able to "save" the coverage achieved in each test session, and subsequently merge the results together. This can be achieved with the `write <dir>` and `merge <dir>` options to the coverage command. The write option saves the current coverage information in `<dir>` for each specification file loaded; the merge option reads this information back, and merges it with the current coverage information. For example, each day's test coverage could be written to a separate "day" directory, and then all the days merged together for review of the overall coverage at the end.

latex|latexdoc [<files>]: This command generates LaTeX coverage files. These are LaTeX versions of the source files with parts of the specification highlighted where they have not been executed. The LaTeX output also contains a table of percentage cover by module/-class and the number of times functions and operations were hit during the execution. The `latexdoc` command is the same, except that output files are wrapped in LaTeX document headers. The output files are written to the same directory as the source files, one per source file, with the extension `.tex`. Coverage information is reset when a specification is loaded, when an `init` command is given, or when the command `coverage clear` is executed, otherwise coverage is cumulative. If several files are loaded, the coverage for just one source file can be listed with `coverage <file>` or `latex <file>`.

files: This command lists the names of all source files loaded.

reload: This command will reload, parse and type check the VDM model files currently loaded. Note that if there are any errors in the parse or type check of the files, the interpreter will exit after the reload.

load <files>: This command replaces the current loaded VDM model files. Note that if there are any errors in the parse or type check of the files, the interpreter will exit after the load.

[q]uit: This command leaves the interpreter.

When the execution of a VDM model is stopped at a breakpoint, there are additional commands that can be used. These are:

[s]tep: This command steps forward until the current expression/statement is on a new line. The command will step into function and operation calls.

[n]ext: This command is similar to step except function and operation calls are stepped over.

[o]ut: This command runs to the return of the current function or operation.

[c]ontinue: This command resumes execution and continues until the next breakpoint or completion of the thread that is being debugged.

stack: This command displays the current stack frame context (i.e. the call stack).

up: This command moves the stack frame context up one frame to allow variables to be seen.

down: This command moves the stack frame context down one frame.

source: This command lists VDM source around the current breakpoint.

stop: This command terminates the execution immediately.

threads: This command can only be used for the VDM++ and VDM-RT dialects. It lists the active threads with status information for each thread.

References

[APIMan]   The VDM Tool Group. *VDM Toolbox API*. Technical Report, CSK Systems, January 2008.

[Battle10]   Nick Battle. VDMJ Design Specification. Available from the Overture Source-Forge repository, September 2010. 59 pages. .

[Bjørner&78a]   D. Bjørner and C.B. Jones, editors. *The Vienna Development Method: The Meta-Language*. Volume 61 of *Lecture Notes in Computer Science*, Springer-Verlag, 1978.

              This was the first monograph on *Meta-IV*. See also entries: [Bjørner78b], [Bjørner78c], [Lucas78], [Jones78a], [Jones78b], [Henhapl&78]

[Bjørner78b]   D. Bjørner. Programming in the Meta-Language: A Tutorial. *The Vienna Development Method: The Meta-Language*, 24–217, 1978.

              An informal introduction to *Meta-IV*

[Bjørner78c]   D. Bjørner. Software Abstraction Principles: Tutorial Examples of an Operating System Command Language Specification and a PL/I-like On-Condition Language Definition. *The Vienna Development Method: The Meta-Language*, 337–374, 1978.

              Exemplifies so called **exit** semantics uses of *Meta-IV* to slightly non-trivial examples.

[Clement&99]   Tim Clement and Ian Cottam and Peter Froome and Claire Jones. The Development of a Commercial "Shrink-Wrapped Application" to Safety Integrity Level 2: the DUST-EXPERT Story. In *Safecomp'99*, Springer Verlag, Toulouse, France, September 1999. LNCS 1698, ISBN 3-540-66488-2.

[DLMan]   The VDM Tool Group. *The Dynamic Link Facility*. Technical Report, CSK Systems, January 2008.

[Elmstrøm&94]   René Elmstrøm and Peter Gorm Larsen and Poul Bøgh Lassen. The IFAD VDM-SL Toolbox: A Practical Approach to Formal Specifications. *ACM Sigplan Notices*, 29(9):77–80, September 1994. 4 pages.

[Fitzgerald&05]   John Fitzgerald and Peter Gorm Larsen and Paul Mukherjee and Nico Plat and Marcel Verhoef. *Validated Designs for Object–oriented Systems*. Springer, New York, 2005.

[Fitzgerald&08a] J. S. Fitzgerald and P. G. Larsen and M. Verhoef. Vienna Development Method. *Wiley Encyclopedia of Computer Science and Engineering*, 2008. 11 pages. edited by Benjamin Wah, John Wiley & Sons, Inc.

[Fitzgerald&08b] John Fitzgerald and Peter Gorm Larsen and Shin Sahara. VDMTools: Advances in Support for Formal Modeling in VDM. *ACM Sigplan Notices*, 43(2):3–11, February 2008. 8 pages.

[Fitzgerald&09]   John Fitzgerald and Peter Gorm Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, Second edition, 2009. ISBN 0-521-62348-0.

[Fitzgerald&98]   John Fitzgerald and Peter Gorm Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 1998. ISBN 0-521-62348-0.

[Henhapl&78]   W. Henhapl, C.B. Jones. A Formal Definition of ALGOL 60 as described in the 1975 modified Report. In *The Vienna Development Method: The Meta-Language*, pages 305–336, Springer-Verlag, 1978.

                  One of several examples of ALGOL 60 descriptions.

[ISOVDM96]   Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language. December 1996.

[Johnson96]   C.W. Johnson. Literate Speifications. *Software Engineering Journal*, 225–237, July 1996.

[Jones78a]   C.B. Jones. The Meta-Language: A Reference Manual. In *The Vienna Development Method: The Meta-Language*, pages 218–277, Springer-Verlag, 1978.

[Jones78b]   C.B. Jones. The Vienna Development Method: Examples of Compiler Development. In Amirchachy and Neel, editors, *Le Point sur la Compilation*, INRIA Publ. Paris, 1979.

[Jones90]   Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, Englewood Cliffs, New Jersey, second edition, 1990. 333 pages. ISBN 0-13-880733-7.

               This book deals with the Vienna Development Method. The approach explains formal (functional) specifications and verified design with an emphasis on the study of proofs in the development process.

[Kurita&09]   T. Kurita and Y. Nakatsugawa. The Application of VDM++ to the Development of Firmware for a Smart Card IC Chip. *Intl. Journal of Software and Informatics*, 3(2-3), October 2009.

[Larsen01]   Peter Gorm Larsen. Ten Years of Historical Development: "Bootstrapping" VDMTools. *Journal of Universal Computer Science*, 7(8):692–709, 2001.

               | http://www.jucs.org/jucs_7_8/ten_years_of_historical—

[Larsen&09]   Peter Gorm Larsen and John Fitzgerald and Sune Wolff. Methods for the Development of Distributed Real-Time Systems using VDM. *International Journal of Software and Informatics*, 3(2-3), October 2009.

[Larsen&10]   Peter Gorm Larsen and Nick Battle and Miguel Ferreira and John Fitzgerald and Kenneth Lausdahl and Marcel Verhoef. The Overture Initiative – Integrating Tools for VDM. *ACM Software Engineering Notes*, 35(1):, January 2010. 6 pages.

[Larsen&95]   Peter Gorm Larsen and Bo Stig Hansen. Semantics for Underdetermined Expressions. *Accepted for "Formal Aspects of Computing"*, 7(??):??, January 1995. 14 pages.

[Lucas78]   P. Lucas. On the Formalization of Programming Languages: Early History and Main Approaches. In *The Systematic Development of Compiling Algorithm*, INRIA Publ. Paris, 1978.

               An historic overview of the (VDL and other) background for VDM.

[Mukherjee&00]   Paul Mukherjee and Fabien Bousquet and Jérôme Delabre and Stephen Paynter and Peter Gorm Larsen. Exploring Timing Properties Using VDM++ on an Industrial Application. In J.C. Bicarregui and J.S. Fitzgerald, editors, *Proceedings of the Second VDM Workshop*, September 2000. Available at www.vdmportal.org.

[Verhoef&06]   Marcel Verhoef and Peter Gorm Larsen and Jozef Hooman. Modeling and Validating Distributed Embedded Real-Time Systems with VDM++. In Jayadev Misra and Tobias Nipkow and Emil Sekerinski, editors, *FM 2006: Formal Methods*, pages 147–162, Lecture Notes in Computer Science 4085, 2006.

Appendix A

Templates in Overture

Overture defines a number of standard Eclipse templates. You can add your own as well. The keys and descriptions of the pre-defined templates are:

Key	Description
caseExpression	Case Expression
dclStatement	Declare
defExpression	def pattern = expression1 in expression2
exists	exists bindList & predicate
forall	forall bind list & predicate
forallLoop	for identifier = expression1 to expression2 do statement
forallinset	forall in set
functions	Function block
ifthen	if predicate then expression1 else expression2
let	let pattern = expression1 in expression2
operations	Operation block
while	while predicate do statement
functionExplicit	Explicit function
functionImplicit	Implicit function
module	Module
moduleSkeleton	Module Full skeleton of a module
operationExplicit	Explicit Operation
operationImplicit	Implicit operation
act	The number of times that operation name operation has been activated
active	The number of operation name operations that are currently active.
class	Class Definition
classSkeleton	Class Definition full skeleton

fin	The number of times that the operation name operation has been completed
functionExplicit	Explicit function
functionImplicit	Implicit function
instancevariables	Instance Variables block
isnotyetspecified	is not yet specified
isofbaseclass	Test if an object is of a specific base class
isofclass	Test if an object is of class
issubclassof	Is subclass of
issubclassresponsibility	Is subclass responsibility
mutex	Mutex operation
operationExplicit	Explicit Operation
operationImplicit	Implicit operation
per	Permission predicate for an operation, history counters can be used: #fin, #act, #active, #req, #waiting
req	The number of requests that has been issued for the operation name operation
samebaseclass	Test if two objects are of the same type
self	Get a reference to the current object
sync	Synchronization block
values	Values block
waiting	The number of outstanding requests for the operation name operation
act	The number of times that operation name operation has been activated
active	The number of operation name operations that are currently active.
bus	BUS (Priority <CSMACD>, capacity,set of connected CPUs)
class	Class Definition
classSkeleton	Class Definition full skeleton
cpu	CPU (Priority <FP/FCFS>, capacity)
cycle	Cycles (number of cycles) statement
duration	Duration (time in nanoseconds) statement
fin	The number of times that the operation name operation has been completed
functionExplicit	Explicit function
functionImplicit	Implicit function
instancevariables	Instance Variables block
isnotyetspecified	is not yet specified
isofbaseclass	Test if an object is of a specific base class
isofclass	Test if an object is of class

issubclassof	Is subclass of
issubclassresponsibility	Is subclass responsibility
mutex	Mutex operation
operationExplicit	Explicit Operation
operationImplicit	Implicit operation
per	Permission predicate for an operation, history counters can be used: #fin, #act, #active, #req, #waiting
periodic	periodic(periode,jitter,delay,offset)(operation name)
req	The number of requests that has been issued for the operation name operation
samebaseclass	Test if two objects are of the same type
self	Get a reference to the current object
sync	Synchronization block
system	System skeleton
time	Get the current time
values	Values block
waiting	The number of outstanding requests for the operation name operation

Appendix B

Internal Errors

This appendix gives a list of the internal errors in Overture and the circumstances under which each internal error can be expected. Most of these errors should *never* be seen, so if they appear please report the occurrence via the Overture bug reporting utility (`https://sourceforge.net/tracker/?group_id=141350&atid=749152`).

0000: `File IO errors, eg. "File not found"` This typically occurs if a specification file is no longer present.

0001: `"Mark/reset not supported - use push/pop"`

0002: `"Cannot change type qualifier: <name><qualifiers> to <qualifiers>"`

0003: `"PatternBind passed <class name>"`

0004: `"Cannot get bind values for type <type>"`

0005: `"Illegal clone"`

0006: `"Constructor for <class> can't find <member>"`

0007: `"Cannot write to IO file <name>"`

0009: `"Too many syntax errors"` This error typically occurs if one have included a file that is in a non VDM format and by mistake have given it a vdm file extension (`vdmsl`, `vdmpp` or `vdmrt`).

0010: `"Too many type checking errors"`

0011: `"CPU or BUS creation failure"`

0052: `"Cannot set default name at breakpoint"`

0053: `"Unknown trace reduction type"`

0054: "Cannot instantiate native object: <reason>"

0055: "Cannot access native object: <reason>"

0056: "Native method cannot use pattern arguments: <sig>"

0057: "Native member not found: <name>"

0058: "Native method does not return Value: "

0059: "Failed in native method: <reason>"

0060: "Cannot access native method: <reason>"

0061: "Cannot find native method: <reason>"

0062: "Cannot invoke native method: <reason>"

0063: "No delegate class found: <name>"

0064: "Native method should be static: <name>"

0065: "Illegal Lock state"

0066: "Thread is not running on a CPU"

Appendix C

Lexical Errors

When a VDM model is parsed, the first phase is to gather the single characters into tokens that can be used in the further processing. This is called a lexical analysis and errors in this area can be as follows:

1000: `"Malformed quoted character"`

1001: `"Invalid char <ch> in base <n> number"`

1002: `"Expecting '|->'"`

1003: `"Expecting '...'"`

1004: `"Expecting '<-:'"`

1005: `"Expecting close double quote"`

1006: `"Expecting close quote after character"`

1007: `"Unexpected tag after '#'"`

1008: `"Malformed module'name"`

1009: `"Unexpected character 'c'"`

1010: `"Expecting <digits>[.<digits>][e<+-><digits>]"`

1011: `"Unterminated block comment"`

Appendix D

Syntax Errors

If the syntax of the file you have provided does not meet the syntax rules for the VDM dialect you wish to use, syntax errors will be reported. These can be as follows:

2000: `"Expecting 'in set' after pattern in set binding"`

2001: `"Expecting 'in set' in set bind"`

2002: `"Expecting ':'  in type bind"`

2003: `"Expecting 'in set' after pattern in binding"`

2004: `"Expecting 'in set' or ':'  after patterns"`

2005: `"Expecting list of 'class' or 'system' definitions"`

2006: `"Found tokens after class definitions"`

2007: `"Expecting 'end <class>'"`

2008: `"Class does not start with 'class'"`

2009: `"Can't have instance variables in VDM-SL"`

2010: `"Can't have a thread clause in VDM-SL"`

2011: `"Only one thread clause permitted per class"`

2012: `"Can't have a sync clause in VDM-SL"`

2013: `"Expected 'operations', 'state', 'functions', 'types' or 'values'"`

2014: `"Recursive type declaration"` This is reported in type definitions such as `T = T`.

2015: `"Expecting =<type> or ::<field list>"`

2016: `"Function name cannot start with 'mk_'"`

2017: `"Expecting ':'  or '(' after name in function definition"`

2018: `"Function type is not a -> or +> function"`

2019: `"Expecting identifier <name> after type in definition"`

2020: `"Expecting '(' after function name"`

2021: `"Expecting ':'  or '(' after name in operation definition"`

2022: `"Expecting name <name> after type in definition"`

2023: `"Expecting '(' after operation name"`

2024: `"Expecting external declarations after 'ext'"`

2025: `"Expecting <name>: exp->exp in errs clause"`

2026: `"Expecting 'rd' or 'wr' after 'ext'"`

2027: `"Expecting +ive number in periodic statement"`

2028: `"Expecting 'per' or 'mutex'"`

2029: `"Expecting <set bind> = <expression>"`

2030: `"Expecting simple field identifier"`

2031: `"Expecting field number after .#"`

2032: `"Expecting field name"`

2033: `"Expected 'is not specified' or 'is subclass responsibility'"`

2034: `"Unexpected token in expression"`

2035: `"Tuple must have >1 argument"`

2036: `"Expecting mk_<type>"`

2037: `"Malformed mk_<type> name <name>"`

2038: `"Expecting is_<type>"`

2039: `"Expecting maplet in map enumeration"`

2040: `"Expecting 'else' in 'if' expression"`

2041: "Expecting two arguments for 'isofbase'"

2042: "Expecting (<class>,<exp>) arguments for 'isofbase'"

2043: "Expecting two arguments for 'isofclass'"

2044: "Expecting (<class>,<exp>) arguments for 'isofclass'"

2045: "Expecting two expressions in 'samebaseclass'"

2046: "Expecting two expressions in 'sameclass'"

2047: "Can't use history expression here"

2048: "Expecting #act, #active, #fin, #req or #waiting"

2049: "Expecting 'end <module>'"

2050: "Expecting library name after 'uselib'"

2051: "Expecting 'end <module>'"

2052: "Expecting 'all', 'types', 'values', 'functions' or 'operations'"

2053: "Exported function is not a function type"

2054: "Expecting types, values, functions or operations"

2055: "Imported function is not a function type"

2056: "Cannot use module'id name in patterns"

2057: "Unexpected token in pattern"

2058: "Expecting identifier"

2059: "Expecting a name"

2060: "Found qualified name <name>. Expecting an identifier"

2061: "Expecting a name"

2062: "Expected 'is not specified' or 'is subclass responsibility'"

2063: "Unexpected token in statement"

2064: "Expecting <object>.identifier(args) or name(args)"

2065: "Expecting <object>.name(args) or name(args)"

2066: "Expecting object field name"

2067: "Expecting 'self', 'new' or name in object designator"

2068: "Expecting field identifier"

2069: "Expecting <identifier>:<type> := <expression>"

2070: "Function type cannot return void type"

2071: "Expecting field identifier before ':'"

2072: "Expecting field name before ':-'"

2073: "Duplicate field names in record type"

2074: "Unexpected token in type expression"

2075: "Expecting 'is subclass of'"

2076: "Expecting 'is subclass of'"

2077: "Expecting 'end' after class members"

2078: "Missing ';' after type definition"

2079: "Missing ';' after function definition"

2080: "Missing ';' after state definition"

2081: "Missing ';' after value definition"

2082: "Missing ';' after operation definition"

2083: "Expecting 'instance variables'"

2084: "Missing ';' after instance variable definition"

2085: "Missing ';' after thread definition"

2086: "Missing ';' after sync definition"

2087: "Expecting '==' after pattern in invariant"

2088: "Expecting '@' before type parameter"

2089: "Expecting '@' before type parameter"

2090: "Expecting ']' after type parameters"

2091: "Expecting ')' after function parameters"

2092: "Expecting '==' after parameters"

2093: "Missing colon after pattern/type parameter"

2094: "Missing colon in identifier/type return value"

2095: "Implicit function must have post condition"

2096: "Expecting <pattern>[:<type>]=<exp>"

2097: "Expecting 'of' after state name"

2098: "Expecting '==' after pattern in invariant"

2099: "Expecting '==' after pattern in initializer"

2100: "Expecting 'end' after state definition"

2101: "Expecting ')' after operation parameters"

2102: "Expecting '==' after parameters"

2103: "Missing colon after pattern/type parameter"

2104: "Missing colon in identifier/type return value"

2105: "Implicit operation must define a post condition"

2106: "Expecting ':' after name in errs clause"

2107: "Expecting '->' in errs clause"

2108: "Expecting <pattern>=<exp>"

2109: "Expecting <type bind>=<exp>"

2110: "Expecting <pattern> in set <set exp>"

2111: "Expecting <pattern> in set <set exp>"

2112: "Expecting '(' after periodic"

2113: "Expecting ')' after period arguments"

2114: "Expecting '(' after periodic(...)"

2115: "Expecting (name) after periodic(...)"

2116: "Expecting <name> => <exp>"

2117: "Expecting '(' after mutex"

2118: "Expecting ')' after 'all'"

2119: "Expecting ')'"

2120: "Expecting 'e1,...,e2' in subsequence"

2121: "Expecting ')' after subsequence"

2122: "Expecting ')' after function args"

2123: "Expecting ']' after function instantiation"

2124: "Expecting ')'"

2125: "Expecting 'is not yet specified"

2126: "Expecting 'is not yet specified"

2127: "Expecting 'is subclass responsibility'"

2128: "Expecting comma separated record modifiers"

2129: "Expecting <identifier> |-> <expression>"

2130: "Expecting ')' after mu maplets"

2131: "Expecting ')' after mk_ tuple"

2132: "Expecting is_(expression, type)"

2133: "Expecting ')' after is_ expression"

2134: "Expecting pre_(function [,args])"

2135: "Expecting '}' in empty map"

2136: "Expecting '}' after set comprehension"

2137: "Expecting 'e1,...,e2' in set range"

2138: "Expecting '}' after set range"

2139: "Expecting '}' after set enumeration"

2140: "Expecting '}' after map comprehension"

2141: "Expecting '}' after map enumeration"

2142: "Expecting ']' after list comprehension"

2143: "Expecting ']' after list enumeration"

2144: "Missing 'then'"

2145: "Missing 'then' after 'elseif'"

2146: "Expecting ':' after cases expression"

2147: "Expecting '->' after others"

2148: "Expecting 'end' after cases"

2149: "Expecting '->' after case pattern list"

2150: "Expecting 'in' after local definitions"

2151: "Expecting 'st' after 'be' in let expression"

2152: "Expecting 'in' after bind in let expression"

2153: "Expecting '&' after bind list in forall"

2154: "Expecting '&' after bind list in exists"

2155: "Expecting '&' after single bind in exists1"

2156: "Expecting '&' after single bind in iota"

2157: "Expecting '&' after bind list in lambda"

2158: "Expecting 'in' after equals definitions"

2159: "Expecting '(' after new class name"

2160: "Expecting '(' after 'isofbase'"

2161: "Expecting ')' after 'isofbase' args"

2162: "Expecting '(' after 'isofclass'"

2163: "Expecting ')' after 'isofclass' args"

2164: "Expecting '(' after 'samebaseclass'"

2165: "Expecting ')' after 'samebaseclass' args"

2166: `"Expecting '(' after 'sameclass'"`

2167: `"Expecting ')' after 'sameclass' args"`

2168: `"Expecting <#op>(name(s))"`

2169: `"Expecting <#op>(name(s))"`

2170: `"Expecting 'module' at module start"`

2171: `"Expecting 'end' after module definitions"`

2172: `"Expecting 'dlmodule' at module start"`

2173: `"Expecting 'end' after dlmodule definitions"`

2174: `"Malformed imports?  Expecting 'exports' section"`

2175: `"Expecting ':'  after export name"`

2176: `"Expecting ':'  after export name"`

2177: `"Expecting ':'  after export name"`

2178: `"Expecting 'imports'"`

2179: `"Expecting 'from' in import definition"`

2180: `"Mismatched brackets in pattern"`

2181: `"Mismatched braces in pattern"`

2182: `"Mismatched square brackets in pattern"`

2183: `"Expecting '(' after mk_ tuple"`

2184: `"Expecting ')' after mk_ tuple"`

2185: `"Expecting '(' after <type> record"`

2186: `"Expecting ')' after <type> record"`

2187: `"Expecting 'is not yet specified"`

2188: `"Expecting 'is not yet specified"`

2189: `"Expecting 'is subclass responsibility'"`

2190: `"Expecting 'exit'"`

2191: "Expecting 'tixe'"

2192: "Expecting '{' after 'tixe'"

2193: "Expecting '|->' after pattern bind"

2194: "Expecting 'in' after tixe traps"

2195: "Expecting 'trap'"

2196: "Expecting 'with' in trap statement"

2197: "Expecting 'in' in trap statement"

2198: "Expecting 'always'"

2199: "Expecting 'in' after 'always' statement"

2200: "Expecting '||'"

2201: "Expecting '(' after '||'"

2202: "Expecting ')' at end of '||' block"

2203: "Expecting 'atomic'"

2204: "Expecting '(' after 'atomic'"

2205: "Expecting ')' after atomic assignments"

2206: "Expecting '(' after call operation name"

2207: "Expecting '(' after new class name"

2208: "Expecting 'while'"

2209: "Expecting 'do' after while expression"

2210: "Expecting 'for'"

2211: "Expecting 'in set' after 'for all'"

2212: "Expecting 'in set' after 'for all'"

2213: "Expecting 'do' after for all expression"

2214: "Expecting 'in' after pattern bind"

2215: "Expecting 'do' before loop statement"

2216: "Expecting '=' after for variable"

2217: "Expecting 'to' after from expression"

2218: "Expecting 'do' before loop statement"

2219: "Missing 'then'"

2220: "Missing 'then' after 'elseif' expression"

2221: "Expecting ':=' in object assignment statement"

2222: "Expecting ':=' in state assignment statement"

2223: "Expecting ')' after map/seq reference"

2224: "Expecting statement block"

2225: "Expecting ';' after statement"

2226: "Expecting ')' at end of statement block"

2227: "Expecting ';' after declarations"

2228: "Expecting name:type in declaration"

2229: "Expecting 'return'"

2230: "Expecting 'let'"

2231: "Expecting 'in' after local definitions"

2232: "Expecting 'st' after 'be' in let statement"

2233: "Expecting 'in' after bind in let statement"

2234: "Expecting 'cases'"

2235: "Expecting ':' after cases expression"

2236: "Expecting '->' after case pattern list"

2237: "Expecting '->' after others"

2238: "Expecting 'end' after cases"

2239: "Expecting 'def'"

2240: "Expecting 'in' after equals definitions"

2241: "Expecting '['"

2242: "Expecting ']' after specification statement"

2243: "Expecting 'start'"

2244: "Expecting 'start('"

2245: "Expecting ')' after start object"

2246: "Expecting 'startlist'"

2247: "Expecting 'startlist('"

2248: "Expecting ')' after startlist objects"

2249: "Missing 'of' in compose type"

2250: "Missing 'end' in compose type"

2251: "Expecting 'to' in map type"

2252: "Expecting 'to' in inmap type"

2253: "Expecting 'of' after set"

2254: "Expecting 'of' after seq"

2255: "Expecting 'of' after seq1"

2256: "Bracket mismatch"

2257: "Missing close bracket after optional type"

2258: "Expecting '==>' in explicit operation type"

2259: "Operations cannot have [@T] type parameters"

2260: "Module starts with 'class' instead of 'module'"

2261: "Missing comma between return types?"

2262: "Can't have traces in VDM-SL"

2263: "Missing ';' after named trace definition"

2264: "Expecting ':' after trace name"

2265: "Expecting 'n1, n2' after trace definition"

2266: "Expecting 'n' or 'n1, n2' after trace definition"

2267: "Expecting 'obj.op(args)' or 'op(args)'"

2268: "Expecting 'id.id(args)'"

2269: "Expecting '(trace definitions)'"

2270: "Only value definitions allowed in traces"

2271: "Expecting 'duration'"

2272: "Expecting 'duration('"

2273: "Expecting ')' after duration"

2274: "Expecting 'cycles'"

2275: "Expecting 'cycles('"

2276: "Expecting ')' after cycles"

2277: "Can't have state in VDM++"

2278: "Async only permitted for operations"

2279: "Invalid breakpoint hit condition"

2280: "System class cannot be a subclass"

2290: "System class can only define instance variables and a constructor"

2291: "'reverse' not available in VDM classic"

2292: "Expecting '|| (...)'"

2293: "Expecting '|| (a, b ,...)'"

2294: "Expecting ')' ending || clause"

2295: "Can't use old name here"

Appendix E

Type Errors and Warnings

If the syntax rules are satisfied, it is still possible to get errors from the type checker. The errors can be as follows:

3000: "Expression does not match declared type"

3001: "Class inherits thread definition from multiple supertypes"

3002: "Circular class hierarchy detected: <name>"

3003: "Undefined superclass: <supername>"

3004: "Superclass name is not a class: <supername>"

3005: "Overriding a superclass member of a different kind: <member>"

3006: "Overriding definition reduces visibility" This error message typically are caused by using a more restrictive access modifier (or none which is interpreted as private) at this place compared to for example an inherited definition.

3007: "Overriding member incompatible type: <member>"

3008: "Overloaded members indistinguishable: <member>"

3009: "Circular class hierarchy detected: <class>"

3010: "Name <name> is ambiguous"

3011: "Name <name> is multiply defined in class"

3012: "Type <name> is multiply defined in class"

3013: "Class invariant is not a boolean expression"

3014: "Expression is not compatible with type bind"

3015: `"Set bind is not a set type?"`

3016: `"Expression is not compatible with set bind"`

3017: `"Duplicate definitions for <name>"`

3018: `"Function returns unexpected type"`

3019: `"Function parameter visibility less than function definition"`
This error message typically are caused by using a more restrictive access modifier (or none which is interpreted as private) at this place compared to for example an inherited definition.

3020: `"Too many parameter patterns"`

3021: `"Too few parameter patterns"`

3022: `"Too many curried parameters"`

3023: `"Too many parameter patterns"`

3024: `"Too few parameter patterns"`

3025: `"Constructor operation must have return type <class>"`

3026: `"Constructor operation must have return type <class>"`

3027: `"Operation returns unexpected type"`

3028: `"Operation parameter visibility less than operation definition"`
This error message typically are caused by using a more restrictive access modifier (or none which is interpreted as private) at this place compared to for example an inherited definition.

3029: `"Function returns unexpected type"`

3030: `"Function parameter visibility less than function definition"`
This error message typically are caused by using a more restrictive access modifier (or none which is interpreted as private) at this place compared to for example an inherited definition.

3031: `"Unknown state variable <name>"`

3032: `"State variable <name> is not this type"`

3033: `"Polymorphic function has not been instantiated: <name>"`

3034: `"Function is already instantiated: <name>"`

3035: `"Operation returns unexpected type"`

3036: "`Operation parameter visibility less than operation definition`"
This error message typically are caused by using a more restrictive access modifier (or none which is interpreted as private) at this place compared to for example an inherited definition.

3037: "`Static instance variable is not initialized: <name>`"

3038: "`<name> is not an explicit operation`"

3039: "`<name> is not in scope`"

3040: "`Cannot put mutex on a constructor`"

3041: "`Duplicate mutex name`"

3042: "`<name> is not an explicit operation`"

3043: "`<name> is not in scope`"

3044: "`Duplicate permission guard found for <name>`"

3045: "`Cannot put guard on a constructor`"

3046: "`Guard is not a boolean expression`"

3047: "`Only one state definition allowed per module`"

3048: "`Expression does not return a value`"

3049: "`Thread statement/operation must not return a value`"

3050: "`Type <name> is infinite`"

3051: "`Expression does not match declared type`"

3052: "`Value type visibility less than value definition`" This error message typically are caused by using a more restrictive access modifier (or none which is interpreted as private) at this place compared to for example an inherited definition.

3053: "`Argument of 'abs' is not numeric`"

3054: "`Type <name> cannot be applied`"

3055: "`Sequence selector must have one argument`"

3056: "`Sequence application argument must be numeric`"

3057: "`Map application must have one argument`"

3058: "`Map application argument is incompatible type`"

3059: `"Too many arguments"`

3060: `"Too few arguments"`

3061: `"Inappropriate type for argument <n>"`

3062: `"Too many arguments"`

3063: `"Too few arguments"`

3064: `"Inappropriate type for argument <n>"`

3065: `"Left hand of <operator> is not <type>"`

3066: `"Right hand of <operator> is not <type>"`

3067: `"Argument of 'card' is not a set"`

3068: `"Right hand of map 'comp' is not a map"`

3069: `"Domain of left should equal range of right in map 'comp'"`

3070: `"Right hand of function 'comp' is not a function"`

3071: `"Left hand function must have a single parameter"`

3072: `"Right hand function must have a single parameter"`

3073: `"Parameter of left should equal result of right in function 'comp'"`

3074: `"Left hand of 'comp' is neither a map nor a function"`

3075: `"Argument of 'conc' is not a seq of seq"`

3076: `"Argument of 'dinter' is not a set of sets"`

3077: `"Merge argument is not a set of maps"`

3078: `"dunion argument is not a set of sets"`

3079: `"Left of '<-:' is not a set"`

3080: `"Right of '<-:' is not a map"`

3081: `"Restriction of map should be set of <type>"`

3082: `"Left of '<:' is not a set"`

3083: `"Right of '<:' is not a map"`

3084: "Restriction of map should be set of <type>"

3085: "Argument of 'elems' is not a sequence"

3086: "Else clause is not a boolean"

3087: "Left and right of '=' are incompatible types"

3088: "Predicate is not boolean"

3089: "Predicate is not boolean"

3090: "Unknown field <name> in record <type>"

3091: "Unknown member <member> of class <class>"

3092: "Inaccessible member <member> of class <class>"

3093: "Field <name> applied to non-aggregate type"

3094: "Field #<n> applied to non-tuple type"

3095: "Field number does not match tuple size"

3096: "Argument to floor is not numeric"

3097: "Predicate is not boolean"

3098: "Function value is not polymorphic"

3099: "Polymorphic function is not in scope"

3100: "Function has no type parameters"

3101: "Expecting <n> type parameters"

3102: "Parameter name <name> not defined"

3103: "Function instantiation does not yield a function"

3104: "Argument to 'hd' is not a sequence"

3105: "<operation> is not an explicit operation"

3106: "<operation> is not in scope"

3107: "Cannot use history of a constructor"

3108: "If expression is not a boolean"

3109: "Argument to 'inds' is not a sequence"

3110: "Argument of 'in set' is not a set"

3111: "Argument to 'inverse' is not a map"

3112: "Iota set bind is not a set"

3113: "Unknown type name <name>"

3114: "Undefined base class type: <class>"

3115: "Undefined class type: <class>"

3116: "Argument to 'len' is not a sequence"

3117: "Such that clause is not boolean"

3118: "Predicate is not boolean"

3119: "Map composition is not a maplet"

3120: "Argument to 'dom' is not a map"

3121: "Element is not of maplet type"

3122: "Argument to 'rng' is not a map"

3123: "Left hand of 'munion' is not a map"

3124: "Right hand of 'munion' is not a map"

3125: "Argument of mk_<type> is the wrong type"

3126: "Unknown type <type> in constructor"

3127: "Type <type> is not a record type"

3128: "Record and constructor do not have same number of fields"

3129: "Constructor field <n> is of wrong type"

3130: "Modifier for <tag> should be <type>"

3131: "Modifier <tag> not found in record"

3132: "mu operation on non-record type"

3133: "Class name <name> not in scope"

3134: `"Class has no constructor with these parameter types"`

3135: `"Class has no constructor with these parameter types"`

3136: `"Left and right of '<>' different types"`

3137: `"Not expression is not a boolean"`

3138: `"Argument of 'not in set' is not a set"`

3139: `"Left hand of <operator> is not numeric"`

3140: `"Right hand of <operator> is not numeric"`

3141: `"Right hand of '++' is not a map"`

3142: `"Right hand of '++' is not a map"`

3143: `"Domain of right hand of '++' must be nat1"`

3144: `"Left of '++' is neither a map nor a sequence"`

3145: `"Argument to 'power' is not a set"`

3146: `"Left hand of <operator> is not a set"`

3147: `"Right hand of <operator> is not a set"`

3148: `"Left of ':->' is not a map"`

3149: `"Right of ':->' is not a set"`

3150: `"Restriction of map should be set of <type>"`

3151: `"Left of ':>' is not a map"`

3152: `"Right of ':>' is not a set"`

3153: `"Restriction of map should be set of <type>"`

3154: `"<name> not in scope"`

3155: `"List comprehension must define one numeric bind variable"`

3156: `"Predicate is not boolean"`

3157: `"Left hand of 'î' is not a sequence"`

3158: `"Right hand of 'î' is not a sequence"`

3159: "Predicate is not boolean"

3160: "Left hand of '\' is not a set"

3161: "Right hand of '\' is not a set"

3162: "Left and right of '\' are different types"

3163: "Left hand of <operator> is not a set"

3164: "Right hand of <operator> is not a set"

3165: "Left and right of intersect are different types"

3166: "Set range type must be an number"

3167: "Set range type must be an number"

3168: "Left hand of <operator> is not a set"

3169: "Right hand of <operator> is not a set"

3170: "Map iterator expects nat as right hand arg"

3171: "Function iterator expects nat as right hand arg"

3172: "'**' expects number as right hand arg"

3173: "First arg of '**' must be a map, function or number"

3174: "Subsequence is not of a sequence type"

3175: "Subsequence range start is not a number"

3176: "Subsequence range end is not a number"

3177: "Left hand of <operator> is not a set"

3178: "Right hand of <operator> is not a set"

3179: "Argument to 'tl' is not a sequence"

3180: "Inaccessible member <name> of class <name>"

3181: "Cannot access <name> from a static context"

3182: "Name <name> is not in scope"

3183: "Exported function <name> not defined in module"

3184: "Exported <name> function type incorrect"

3185: "Exported operation <name> not defined in module"

3186: "Exported operation type does not match actual type"

3187: "Exported type <type> not defined in module"

3188: "Exported value <name> not defined in module"

3189: "Exported type does not match actual type"

3190: "Import all from module with no exports?"

3191: "No export declared for import of type <type> from <module>"

3192: "Type import of <name> does not match export from <module>"

3193: "No export declared for import of value <name> from <module>"

3194: "Type of value import <name> does not match export from <module>"

3195: "Cannot import from self"

3196: "No such module as <module>"

3197: "Expression matching set bind is not a set"

3198: "Type bind not compatible with expression"

3199: "Set bind not compatible with expression"

3200: "Mk_ expression is not a record type"

3201: "Matching expression is not a compatible record type"

3202: "Record pattern argument/field count mismatch"

3203: "Sequence pattern is matched against <type>"

3204: "Set pattern is not matched against set type"

3205: "Matching expression is not a product of cardinality <n>"

3206: "Matching expression is not a set type"

3207: "Object designator is not an object type"

3208: "Object designator is not an object type"

3209: `"Member <field> is not in scope"`

3210: `"Object member is neither a function nor an operation"`

3211: `"Expecting <n> arguments"`

3212: `"Unexpected type for argument <n>"`

3213: `"Operation <name> is not in scope"`

3214: `"Cannot call <name> from static context"`

3215: `"<name> is not an operation"`

3216: `"Expecting <n> arguments"`

3217: `"Unexpected type for argument <n>"`

3218: `"Expression is not boolean"`

3219: `"For all statement does not contain a set type"`

3220: `"From type is not numeric"`

3221: `"To type is not numeric"`

3222: `"By type is not numeric"`

3223: `"Expecting sequence type after 'in'"`

3224: `"If expression is not boolean"`

3225: `"Such that clause is not boolean"`

3226: `"Incompatible types in object assignment"`

3228: `"<name> is not in scope"`

3229: `"<name> should have no parameters or return type"`

3230: `"<name> is implicit"`

3231: `"<name> should have no parameters or return type"`

3232: `"<name> is not an operation name"`

3233: `"Precondition is not a boolean expression"`

3234: `"Postcondition is not a boolean expression"`

3235: "Expression is not a set of object references"

3236: "Class does not define a thread"

3237: "Class does not define a thread"

3238: "Expression is not an object reference or set of object references"

3239: "Incompatible types in assignment"

3241: "Body of trap statement does not throw exceptions"

3242: "Map element assignment of wrong type"

3243: "Seq element assignment is not numeric"

3244: "Expecting a map or a sequence"

3245: "Field assignment is not of a record type"

3246: "Unknown field name, <name>"

3247: "Unknown state variable <name> in assignment"

3248: "Cannot assign to 'ext rd' state <name>"

3249: "Object designator is not a map, sequence, function or operation"

3250: "Map application must have one argument"

3251: "Map application argument is incompatible type"

3252: "Sequence application must have one argument"

3253: "Sequence argument is not numeric"

3254: "Too many arguments"

3255: "Too few arguments"

3256: "Inappropriate type for argument <n>"

3257: "Too many arguments"

3258: "Too few arguments"

3259: "Inappropriate type for argument <n>"

3260: "Unknown class member name, <name>"

3261: "Unknown field name, <name>"

3262: "Field assignment is not of a class or record type"

3263: "Cannot reference 'self' from here"

3264: "At least one bind cannot match set"

3265: "At least one bind cannot match this type"

3266: "Argument is not an object"

3267: "Empty map cannot be applied"

3268: "Empty sequence cannot be applied"

3269: "Ambiguous function/operation name: <name>"

3270: "Measure <name> is not in scope"

3271: "Measure <name> is not an explicit function"

3272: "Measure result type is not a nat, or a nat tuple"

3273: "Measure not allowed for an implicit function"

3274: "External variable is not in scope: <name>"

3275: "Error clause must be a boolean"

3276: "Ambiguous names inherited by <name>"

3277: "Trace repeat illegal values"

3278: "Cannot inherit from system class <name>"

3279: "Cannot instantiate system class <name>"

3280: "Argument to deploy must be an object"

3281: "Arguments to duration must be integer >= 0"

3282: "Arguments to cycles must be integer >= 0"

3283: "System class constructor cannot be implicit"

3284: "System class can only define instance variables and a constructor"

3285: "System class can only define a default constructor"

3286: `"Constructor cannot be 'async'"`

3287: `"Periodic thread must have <n> argument(s)"`

3288: `"Period argument must be non-zero"`

3289: `"Delay argument must be less than the period"`

3290: `"Argument to setPriority must be an operation"`

3291: `"Argument to setPriority cannot be a constructor"`

3292: `"Constructor is not accessible"`

3293: `"Asynchronous operation <name> cannot return a value"`

3294: `"Only one system class permitted"`

3295: `"Argument to 'reverse' is not a sequence"`

3296: `"Cannot use '" + typename + "' outside system class"`

3297: `"Cannot use default constructor for this class"`

3298: `"Cannot inherit from CPU"`

3299: `"Cannot inherit from BUS"`

3300: `"Operation <type> cannot be called from a function"`

3301: `"Variable <name> in scope is not updatable"`

3302: `"Variable <name> cannot be accessed from this context"`

3303: `"Measure parameters different to function"`

3304: `"Recursive function cannot be its own measure"`

3305: `"CPU frequency to slow: <speed> Hz"`

3306: `"CPU frequency to fast: <speed> Hz"`

3307: `"Errs clause is not bool -> bool"`

3308: `"Cannot mix modules and flat specifications"`

Warnings from the type checker include:

5000: `"Definition <name> not used"`

5001: `"Instance variable is not initialized: <name>"`

5002: `"Mutex of overloaded operation"` This warning is provided if one defined a mutex for an operation that is defined using overloading. The users needs to be aware that all of the overloaded operations will now by synchronisation controlled by this constraint.

5003: `"Permission guard of overloaded operation"`

5004: `"History expression of overloaded operation"`

5005: `"Should access member <member> from a static context"`

5006: `"Statement will not be reached"`

5007: `"Duplicate definition: <name>"`

5008: `"<name/location> hides <name/location>"`

5009: `"Empty set used in bind"`

5010: `"State init expression cannot be executed"`

5012: `"Recursive function has no measure"` Whenever a recursive function is defined the user have the possibility defining a measure (i.e. a function that takes the same parameters as the recursive function and returns a natural number that should decrease at every recursive call). If such measures are included the proof obligation generator can provide proof obligations that will ensure termination of the recursion.

5014: `"Uninitialized BUS ignored"` This warning appears if one has defined a BUS that is not used.

5015: `"LaTeX source should start with %comment, \document, \section or \subsection"`

Appendix F

Run-Time Errors

When using the interpreter/debugger it is possible to get run-time errors, even if there are no type checking errors. The possible errors are as follows:

4000: `"Cannot instantiate abstract class <class>"`

4002: `"Expression value is not in set bind"`

4003: `"Value <value> cannot be applied"`

4004: `"No cases apply for <value>"`

4005: `"Duplicate map keys have different values"`

4006: `"Type <type> has no field <field>"`

4007: `"No such field in tuple: #<n>"`

4008: `"No such type parameter @<name> in scope"`

4009: `"Type parameter/local variable name clash, @<name>"`

4010: `"Cannot take head of empty sequence"`

4011: `"Illegal history operator: <#op>"`

4012: `"Cannot invert non-injective map"`

4013: `"Iota selects more than one result"`

4014: `"Iota does not select a result"`

4015: `"Let be st found no applicable bindings"`

4016: `"Duplicate map keys have different values: <domain>"`

4017: "Duplicate map keys have different values: <domain>"

4018: "Maplet cannot be evaluated"

4019: "Sequence cannot extend to key: <index>"

4020: "State value is neither a <type> nor a <type>"

4021: "Duplicate map keys have different values: <key>"

4022: "mk_ type argument is not <type>"

4023: "Mu type conflict? No field tag <tag>"

4024: "'not yet specified' expression reached"

4025: "Map key not within sequence index range: <key>"

4026: "Cannot create post_op environment"

4027: "Cannot create pre_op environment"

4028: "Sequence comprehension pattern has multiple variables"

4029: "Sequence comprehension bindings must be numeric"

4030: "Duplicate map keys have different values: <key>"

4031: "First arg of '**' must be a map, function or number"

4032: "'is subclass responsibility' expression reached"

4033: "Tail sequence is empty"

4034: "Name <name> not in scope"

4035: "Object has no field: <name>"

4036: "ERROR statement reached"

4037: "No such field: <name>"

4038: "Loop, from <value> to <value> by <value> will never terminate"

4039: "Set bind does not contain value <value>"

4040: "Let be st found no applicable bindings"

4041: "'is not yet specified' statement reached"

4042: `"Sequence does not contain key: <key>"`

4043: `"Object designator is not a map, sequence, operation or function"`

4045: `"Object does not contain value for field: <name>"`

4046: `"No such field: <name>"`

4047: `"Cannot execute specification statement"`

4048: `"'is subclass responsibility' statement reached"`

4049: `"Value <value> is not in set bind"`

4050: `"Value <value> is not in set bind"`

4051: `"Cannot apply implicit function: <name>"`

4052: `"Wrong number of arguments passed to <name>"`

4053: `"Parameter patterns do not match arguments"`

4055: `"Precondition failure: <pre_name>"` This error occurs if a pre-condition to a function or operation is violated.

4056: `"Postcondition failure: <post_name>"` This error occurs if a post-condition to a function or operation is violated.

4057: `"Curried function return type is not a function"`

4058: `"Value <value> is not a nat1"`

4059: `"Value <value> is not a nat"`

4060: `"Type invariant violated for <type>"`

4061: `"No such key value in map: <key>"`

4062: `"Cannot convert non-injective map to an inmap"`

4063: `"Duplicate map keys have different values: <domain>"`

4064: `"Value <value> is not a nat1 number"`

4065: `"Value <value> is not a nat"`

4066: `"Cannot call implicit operation: <name>"`

4067: `"Deadlock detected"`

4068: "Wrong number of arguments passed to <name>"

4069: "Parameter patterns do not match arguments"

4071: "Precondition failure: <pre_name>"

4072: "Postcondition failure: <post_name>"

4073: "Cannot convert type parameter value to <type>"

4074: "Cannot convert <value> to <type>"

4075: "Value <value> is not an integer"

4076: "Value <value> is not a nat1"

4077: "Value <value> is not a nat"

4078: "Wrong number of fields for <type>"

4079: "Type invariant violated by mk_ arguments"

4080: "Wrong number of fields for <type>"

4081: "Field not defined: <tag>"

4082: "Type invariant violated by mk_ arguments"

4083: "Sequence index out of range: <index>"

4084: "Cannot convert empty sequence to seq1"

4085: "Cannot convert tuple to <type>"

4086: "Value of type parameter is not a type"

4087: "Cannot convert <value> (<kind>) to <type>"

4088: "Set not permitted for <kind>"

4089: "Can't get real value of <kind>"

4090: "Can't get rat value of <kind>"

4091: "Can't get int value of <kind>"

4092: "Can't get nat value of <kind>"

4093: "Can't get nat1 value of <kind>"

4094: `"Can't get bool value of <kind>"`

4095: `"Can't get char value of <kind>"`

4096: `"Can't get tuple value of <kind>"`

4097: `"Can't get record value of <kind>"`

4098: `"Can't get quote value of <kind>"`

4099: `"Can't get sequence value of <kind>"`

4100: `"Can't get set value of <kind>"`

4101: `"Can't get string value of <kind>"`

4102: `"Can't get map value of <kind>"`

4103: `"Can't get function value of <kind>"`

4104: `"Can't get operation value of <kind>"`

4105: `"Can't get object value of <kind>"`

4106: `"Boolean pattern match failed"`

4107: `"Character pattern match failed"`

4108: `"Sequence concatenation pattern does not match expression"`

4109: `"Values do not match concatenation pattern"`

4110: `"Expression pattern match failed"`

4111: `"Integer pattern match failed"`

4112: `"Quote pattern match failed"`

4113: `"Real pattern match failed"`

4114: `"Record type does not match pattern"`

4115: `"Record expression does not match pattern"`

4116: `"Values do not match record pattern"`

4117: `"Wrong number of elements for sequence pattern"`

4118: `"Values do not match sequence pattern"`

4119: "Wrong number of elements for set pattern"

4120: "Values do not match set pattern"

4121: "Cannot match set pattern"

4122: "String pattern match failed"

4123: "Tuple expression does not match pattern"

4124: "Values do not match tuple pattern"

4125: "Set union pattern does not match expression"

4126: "Values do not match union pattern"

4127: "Cannot match set pattern"

4129: "Exit <value>"

4130: "Instance invariant violated: <inv_op>"

4131: "State invariant violated: <inv_op>"

4132: "Using undefined value"

4133: "Map range is not a subset of its domain: <key>"

4134: "Infinite or NaN trouble"

4135: "Cannot instantiate a system class"

4136: "Cannot deploy to CPU"

4137: "Cannot set operation priority on CPU"

4138: "Cannot set CPU priority for operation"

4139: "Multiple BUS routes between CPUs <name> and <name>"

4140: "No BUS between CPUs <name> and <name>"

4141: "CPU policy does not allow priorities"

4142: "Value already updated by thread <n>"

4143: "No such test number: <n>"

4144: "State init expression cannot be executed"

4145: `"Time: <n> is not a nat1"`

4146: `"Measure failure: f(args), measure <name>, current <value>, previous <value>"`

4147: `"Polymorphic function missing @T"`

4148: `"Measure function is called recursively: <name>"`

4149: `"CPU frequency to slow: <speed> Hz"`

4150: `"CPU frequency to fast: <speed> Hz"`

Appendix G

Categories of Proof Obligations

This appendix provides a list of the different proof obligation categories generated by Overture, and an explanantion of the circumstances under which each category can be expected.

map apply: Whenever a map application is made you need to be certain that the argument is in the domain of the map.

function apply: Whenever a function application is used you need to be certain that the list of arguments to the function satisfies the pre-condition of the function, assuming such a predicate is present.

sequence apply: Whenever a sequence application is used you need to be certain that the argument is within the indices of the sequence.

post condition:

function satisfiability: For all implicit function definitions this proof obligation will be generated to ensure that it is possible to find an implementation satisfying the post-conditions for all arguments satisfying the pre-conditions.

function parameter patterns:

let be st existence: Whenever a let-be-such-that expression/statement is used you need to be certain that at least one value will match the such-that expression.

unique existence binding: The `iota` expression requires one unique binding to be present and that is guranteed by proof obligations from this category.

function iteration:

map iteration:

function compose:

map compose:

non-empty set: This kind of proof obligation is used whenever non-empty sets are required.

non-empty sequence: This kind of proof obligation is used whenever non-empty sequences are required (eg. taking the head of a sequence)

non-zero: This kind of proof obligation is used whenever zero cannot be used (e.g. in division).

finite map: If a type binding to a type that potentially has infinitely many elements is used inside a map comprehension, this proof obligation will be generated because all mappings in VDM must be finite.

finite set: If a type binding to a type that potentially has infinitely many elements is used inside a set comprehension, this proof obligation will be generated because all sets in VDM must be finite.

map compatible: Mappings in VDM represent a unique relationship between the domain values and the corresponding range values. Proof obligations in this category are meant to ensure that such a unique relationship is guranteed.

map sequence compatible:

map set compatible:

sequence modification:

tuple selection: This proof obligation category is used whenever a tuple selection expression is used to gurantee that the length of the tuple is at least as long as the selector used.

value binding:

subtype: This proof obligation category is used whenever it is not possible to statically detect that the given value falls into the subtype required.

cases exhaustive: If a cases expression does not have an `others` clause it is necessary to ensure that the different case alternatives catch all values of the type of the expression used in the case choice.

type invariant: Proof obligations from this category are used to ensure that invariants for elements of a particular type are satisfied.

recursive function: This proof obligation makes use of the `measure` construct to ensure that a recursive function will terminate.

state invariant: If a state (including instance variables in VDM++) has an invariant, this proof obligation will be generated whenever an assignment is made to a part of the state.

while loop termination: This kind of proof obligation is a reminder to ensure that a while loop will terminate.

operation post condition: Whenever an explicit operation has a post-condition there is an implicit proof obligation generated to remind the user that you have to ensure that the explicit body of the operation satisfies the post-condition for all possible inputs.

operation parameter patterns:

operation satifiability: For all implicit operation definitions this proof obligation will be generated to ensure that it is possible to find an implementation satisfying the post-condition for all arguments satisfying the pre-conditions.

Appendix H

Index

Index