

**Overture Technical Report Series
No. TR-002**

September 2016

Overture VDM-10 Tool Support: User Guide

Version 2.4.2

by

Peter Gorm Larsen, Kenneth Lausdahl, Peter Tran-Jørgensen,
Joey Coleman, Sune Wolff, Luís Diogo Couto and Victor Bandur
Aarhus University, Department of Engineering
Finlandsgade 22, DK-8000 Århus C, Denmark

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





Document history

Month	Year	Version	Version of Overture
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
August	2013	4	2.0.0
January	2015	5	2.1.6
April	2015	5	2.2.4
September	2015	6	2.3.0
September	2016	7	2.4.2

Contents

1	Introduction	3
2	Getting Hold of the Software	5
3	Using the VDM Perspective	7
3.1	Understanding Eclipse Terminology	7
3.2	Additional Eclipse Features Applicable in Overture	9
3.2.1	Opening and Closing Projects	9
3.2.2	Adding Additional VDM File Extensions	10
3.2.3	Filtering Project Contents	10
3.2.4	Including line numbers in the Editor	10
4	Managing Overture Projects	13
4.1	Importing Overture Projects	13
4.2	Creating a New Overture Project	14
4.3	Creating Files	14
4.4	Adding Standard Libraries	15
4.5	Setting Project Options	16
5	Editing VDM Models	21
5.1	VDM Dialect Editors	21
5.2	Using Templates	21
6	Interpretation and Debugging in Overture	23
6.1	Run and Debug Launch Configurations	23
6.2	The Debug Perspective	26
6.2.1	The Debug View	27
6.2.2	The Variables View	27
6.2.3	The Breakpoints View	28
6.2.4	Conditional Breakpoints	28
6.2.5	The Expressions View	28
7	Collecting Test Coverage Information	31



8	Pretty Printing to \LaTeX	33
9	Managing Proof Obligations	35
10	Combinatorial Testing	37
10.1	Using the Combinatorial Testing GUI	37
11	Automatic Generation of Code	39
11.1	Use of the Java Code Generator	39
11.2	Configuration of the Java Code Generator	40
11.2.1	Disable cloning	42
11.2.2	Generate character sequences as strings	42
11.2.3	Generate concurrency mechanisms	42
11.2.4	Generate Java Modeling Language (JML) annotations	42
11.2.5	Use JML <code>\invariant_for</code> to explicitly check record invariants	43
11.2.6	Generate VDM location information for code generated constructs	43
11.2.7	Choose output package	43
11.2.8	Skip classes/modules during the code generation process	43
11.3	Limitations of the Java Code Generator	44
11.4	The Code Generation Runtime Library	45
11.5	Translation of the VDM types and type constructors	45
11.6	The C Code Generator	46
11.6.1	Language Feature Support Status	46
11.6.2	The C Translation Strategy	48
12	Mapping VDM++ To and From UML	69
13	Moving from VDM++ to VDM-RT	71
14	Graphics Plotting for VDM Models	73
15	Analysing Logs from VDM-RT Executions	77
16	Expanding VDM Models Scope and Functionality by Linking Java and VDM	81
16.1	Configuring the Java Build Path	81
16.2	Defining Your Own Java Libraries to be used from Overture	82
16.2.1	External Library Example	83
16.3	Enabling Remote Control of the Overture Interpreter	84
16.3.1	Example of a Remote Control Class	84
16.4	Using a GUI in a VDM model: Linking Example	85
16.4.1	The Modelled System	85
16.4.2	External Java Library	86
16.4.3	Remote Control	89



16.4.4	Deployment of the Java Program to Overture	91
16.5	Generated GUI of VDM Models	92
17	A Command-Line Interface to Overture	97
17.1	Starting Overture at the Command-Line	97
17.2	Parsing, Type Checking, and Proof Obligations Command-Line	98
17.3	The Command-Line Interpreter and Debugger	99
	References	109
A	Templates in Overture	111
B	Internal Errors	115
C	Lexical Errors	117
D	Syntax Errors	119
E	Type Errors and Warnings	133
F	Run-Time Errors	149
G	Categories of Proof Obligations	157
H	Mapping Rules between VDM++/VDM-RT and UML Models	161
I	Using VDM Values in Java	165
I.1	The Value Class Hierarchy	165
I.2	Primitive Values	165
I.3	Sets, Sequences and Maps	167
I.4	Other Types	169
I.4.1	Function values	169
I.4.2	Object Values	169
I.4.3	Record Values	170
I.4.4	Token Values	170
I.4.5	Tuple Values	171
I.4.6	Invariant Values	171
I.4.7	Void Values	171



ABSTRACT

This document is the user manual for the Overture Integrated Development Environment (IDE) for the Vienna Development Method (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). The different dialects are controlled by a VDM language Board that evaluates possible Requests for Modifications. Overture tool support is built 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 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 has been used by practitioners who were not specialists in the underlying formalism or logic [Larsen&96, Clement&99, Kurita&09]. Experience with the method suggests that the effort spend on formal modelling and analysis can be recovered in reduced rework costs arising from design errors.

VDM models can be 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 characterise 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]. A further extension to VDM++, called VDM Real Time (VDM-RT¹), includes support for discrete time models [Mukherjee&00, Verhoef&06]. The VDM-RT dialect is also used inside the Crescendo tool² supporting collaborative modelling and co-simulation [Fitzgerald&14]. 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

¹Formerly called VDM In a Constrained Environment (VICE).

²See <http://crescendotool.org/>.



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&13]. It can be seen as an open source version of the closed (but now freely available) tool called VDMTools [Elmstrøm&94,Larsen01,Fitzgerald&08b].

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 15 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 how it is possible automatically to generate executable code in programming languages such as Java and C++ for a subset of VDM models; Chapter 12 explains the support for mapping between object-oriented VDM models and UML models; Chapter 13 shows how a VDM++ project can be converted into a new VDM-RT project; Chapter 14 shows how it is possible to plot values of numeric variables from a model while a VDM model is being interpreted; Chapter 15 shows how to analyse and display execution logs from VDM-RT models; Chapter 16 demonstrates how Java can be used in combination with VDM models and Chapter 17 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. Appendix H provides an overview of the VDM++/VDM-RT to UML mapping rules. Appendix I provides details about how to represent VDM values in order to combine Java with VDM as described in Chapter 16. Finally, there is an index of significant terms used in this manual.

Chapter 2

Getting Hold of the Software

Overture: This is an open source tool, developed by volunteers and built on the Eclipse platform. The project is managed on GitHub¹. The best way to run Overture is to download a special version of Eclipse with the Overture functionality already pre-installed. If you go to:

`http://overturetool.org/download`

you will find pre-installed versions of Overture for Windows, Linux and Mac².

Modelio: In order to take advantage of the UML-VDM mapping, a tool for UML modelling is needed. We recommend Modelio version 3.3.1 since the UML-VDM mapping tool is tested with that version of Modelio. Modelio is both available in a commercial version as well as in an open source setting from a company called Softeam. Just like Overture this tool is built on top of the Eclipse platform. The product can be obtained from

`http://www.modelio.org/`.

Note that in order to be able to execute Overture you need to have Java Runtime Environment (minimum version 1.6) 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*.

Finally, in order to make use of the test coverage feature described in Section 7 it is necessary to have the text processing system called L^AT_EX and its pdf_lat_ex feature. This can for example be obtained from:

- Windows: `http://miktex.org`

¹`https://github.com/overturetool/overture`

²It is planned to develop an update facility, allowing updates to be applied directly from within the Overture tools without requiring a reinstallation. However, this can be a risky process because of the dependencies on non-Overture components and so is not yet supported.



Figure 2.1: The Overture Welcome Screen

- Mac: <http://tug.org/mactex/>
- Linux: Most distributions offer \LaTeX packages

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.

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



Figure 3.3: Icons in the Outline View

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 `.vdmssl`, `.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.

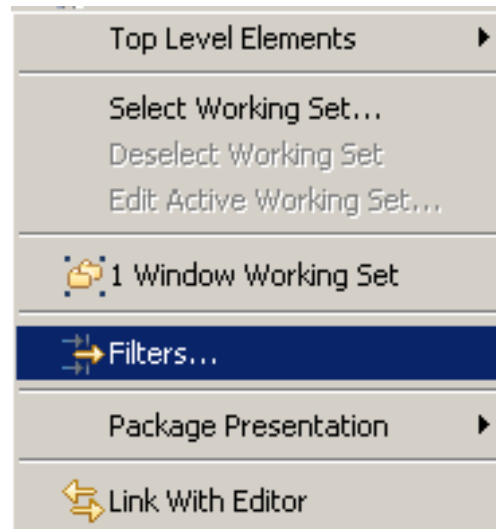


Figure 3.5: Filtering Directories without source files

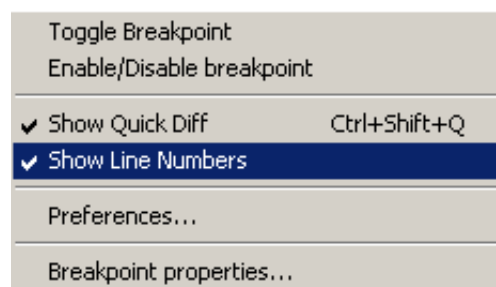


Figure 3.6: Adding Line Numbers in Editor



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*.

It is possible to automatically import a large collection of existing examples. To do this, right click the Explorer view and select *Import* → *Overture* → *Overture Examples*. You can then select which VDM dialect you wish to import examples for. Finally, a selection screen with all examples will be shown¹. Simply pick the ones you wish to import. See Figure 4.1 for more details.



Figure 4.1: Import VDM Examples

¹Note that any previously imported examples will be greyed out.



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);
6. The next step enables references to other projects (but this is not used at the moment);
7. The next step enables inclusion of VDM libraries and
8. Click the `Finish` button (see Figure 4.2).

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 this perspective you can create files using one of the following methods:

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 `.vdmrtd`) 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.



Figure 4.2: Create Project Wizard

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 4.3. 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 separate 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 4.3: Adding New Libraries

All these libraries except `VDM-Unit` are available for all VDM dialects also when a flat VDM-SL specification is used. `VDM-Unit` use object-orientation and thus it cannot be used with VDM-SL.

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 4.4. The options that can be set for each VDM project are:

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



Figure 4.4: Overture Project Settings

necessarily backwards compatible) version of the VDM dialects developed by the Overture VDM 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 4.5. 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`). Finally, the “Model only” option is used to select if you wish to include only the VDM model or also the text that can be written outside `\begin{vdm_al}` and `\end{vdm_al}` environments (see Chapter 8 for more details).

It is also possible to set various preferences that apply to all projects. This is done in the general VDM preferences dialog under *Window* \rightarrow *Preferences* \rightarrow *VDM*. Here, for example, it is possible to link projects to VDMTools if you have the appropriate SCSK VDMTools executables installed on the computer². Figure 4.6 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* \rightarrow *Open project in VDMTools*. Then a project file for VDMTools will automatically be generated with all the files from the Overture project and VDMTools 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.

²This does not work for VDM-RT models since that dialect is no longer supported by VDMTools.

Figure 4.5: Overture Project Settings for \LaTeX

It is also possible to set a few other areas (debugger and dot) but these are mostly used by the developers. Finally it is possible to manage VDM templates, but that is described in Section 5.2.



Figure 4.6: Overture Preferences for connections to VDMTools

In the same fashion it is possible to set preferences for the way VDM++ and VDM-RT models are mapped to UML. This can be seen in Figure 4.7. More information about these preferences can be found in Chapter 12.



Figure 4.7: Overture Preferences for mapping to UML



Chapter 5

Editing VDM Models

5.1 VDM Dialect Editors

VDM model files are always ment to be 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 you are new to 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. In addition to the templates, pressing *CTRL+space* will list completion proposals for types, value constructors and calls to operations and functions. Overture's auto-completion feature is available in versions 2.4.2 onwards, and is similar to that of the Eclipse Java IDE, albeit not as powerful.

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 where to adjust templates 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 chapter 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 an “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 values¹.

However, there are other launch mode possibilities here as well. The “Remote Control” launch mode is advanced but it is explained in more detail in Section 16.3. 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 time². The commands that can be used in the “Console” view correspond to the commands you can give at command-line when it has been started in interpreter mode (see Section 17.3).

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.

¹You 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 ...”

²Those familiar with VDMTools will recognise this functionality 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. By default all possible run-time checks are switched on but if desired (some of) these can be switched 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 here³.

³However a consequence of doing that is that you may get internal Java errors (null pointer or class cast exceptions typically) rather than nice clean VDM type errors about mismatched types.



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 here⁴.

In the launch configuration the “*Debugger*” pane shown in Figure 6.3 can also be useful in rare cases where one has particularly 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 needed.



Figure 6.2: The launch configuration dialog

⁴Note that this feature may not work correctly with the presence of mutually recursive function definitions.



Figure 6.3: The launch configuration dialog

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 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.

The Debug Perspective is illustrated in Figure 6.4.

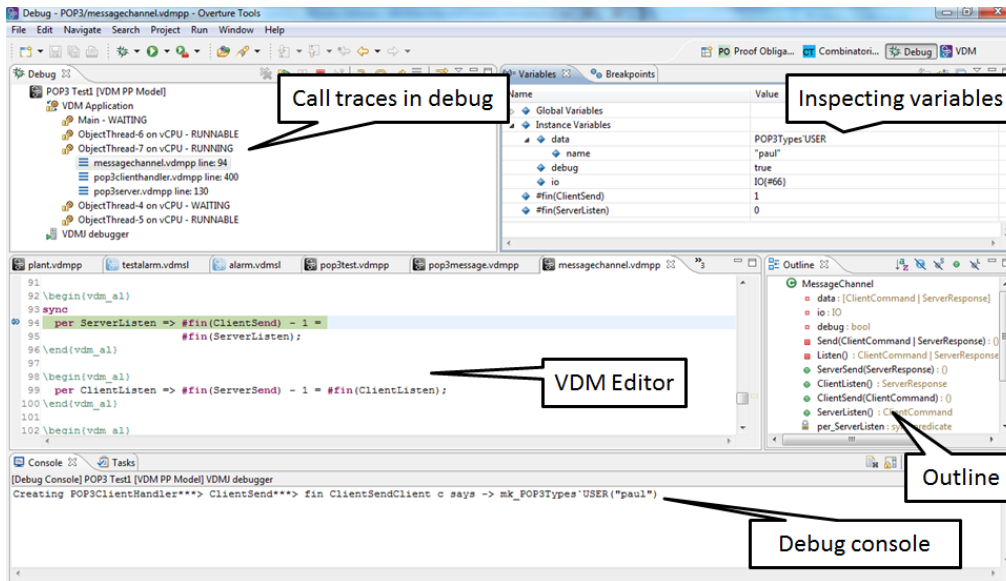


Figure 6.4: Debugging perspective

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.



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

6.2.3 The Breakpoints View

Breakpoints can be added in any perspective from the Editor view⁵. The debug perspective also has a *Breakpoints* view that lists all current breakpoints, allowing you to navigate easily to the location 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 that allows the developer to specify a condition or expression which must be true for the debugger to stop the execution at the given breakpoint. A conditional breakpoint may specify a hit count, and whether the execution should stop at the given breakpoint when the hit count is equal to, greater than, or a multiple of the given value. The condition can also be a simple user-defined expression that is allowed to refer to values and variables that are visible from the given context. The condition can, however, not be an arbitrary expression that contains object or operation calls.

A normal breakpoint can be made conditional by right clicking on the breakpoint mark in the Editor view⁶ 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.

⁵Note that breakpoints can only be set on lines that contain executable code.

⁶Note 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 L^AT_EX

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 L^AT_EX 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 L^AT_EX construct: a `\documentclass`, `\section`, `\subsection` or a L^AT_EX comment.

When using this functionality, it is possible to configure additional options for the PdfLatex generation in the project properties (see Figure 8.1). For example, you can choose whether to use an auto generated main `.tex` file or a user provided one. It is also possible to generate a pdf containing only the model (all text outside `vdm_al` will be ignored).



Figure 8.1: The L^AT_EX project properties window



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. Once you have generated POs for a VDM project for the first time, they will automatically be re-generated whenever the project is rebuilt as long as you stay in the *Proof Obligations* perspective.

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

- A unique number in the list shown;
- The name of the definition in which the proof obligation is located; and
- The proof obligation category (type).



The screenshot displays the Overture VDM-10 tool interface. The top-left pane shows the source code for `plant.vdmp` with variables `alarms` and `schedule`, and a function `PlantInv`. The top-right pane, titled **PO Proof Obligation Explorer**, contains a table of proof obligations. The bottom pane, titled **PO Proof Obligation View**, shows the logical expression for the selected obligation.

No.	PO Name	Type
1	Test1'plant	enumeration map injectivity
2	Plant'PlantInv(set of (Alarm), map (Period) to (set of (Expert)))	legal map application
3	Plant'PlantInv(set of (Alarm), map (Period) to (set of (Expert)))	legal map application
4	Plant'ExpertToPage(Alarm, Period)	let be st existence
5	Plant'ExpertToPage(Alarm, Period)	legal map application
6	Plant'ExpertToPage(Alarm, Period)	legal map application
7	Plant'ExpertToPage(Alarm, Period)	operation establishes postcondition
8	Plant'NumberOfExperts(Period)	legal map application
9	Plant'ExpertIsOnDuty(Expert)	legal map application
10	Plant'Plant(set of (Alarm), map (Period) to (set of (Expert)))	state invariant holds

PO Proof Obligation View

`(forall as:set of (Alarm), sch:map (Period) to (set of (Expert)) & (forall p in set (dom sch) & (p in set (dom sch))))`

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++ (and subsequently VDM-SL and VDM-RT)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 [Larsen&10]. 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 send it to the interpreter for execution (*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.

¹Note that this is only available for VDM-SL and VDM-RT 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

Automatic Generation of Code

It is possible to generate Java code for a large subset of VDM-SL and VDM++ models. In addition to Java, C and C++ code generators are currently being developed. Both these code generators are in the early stages of development. The C++ generator is not included with releases of Overture yet, but the C generator is far enough along that it can be included for testing purposes. For comparison, code generation of VDM-SL and VDM++ specifications to both Java and C++ is a feature that is available in VDMTools [Java2VDMMan, CGMan, CGManPP]. The majority of this chapter focuses solely on the Java code generator available in Overture, with a brief discussion of the C code generator ending the chapter.

11.1 Use of the Java Code Generator

The Java code generator can be launched via the context menu as shown in Figure 11.1. Alternatively, this can be done by highlighting the project in the VDM explorer and typing one of the shortcuts associated to this plugin.

The Java code generator operates in two different modes:

- *Regular mode:* In this mode the Java code generator produces an Eclipse project with all the generated code. Java code generation in this mode can also be initiated using the `Ctrl+Alt+C` shortcut.
- *Launch Configuration mode:* Is currently limited to VDM++. This mode is like regular code generation except that the Java code generator also prompts the user for a launch configuration as input for the code generation process. Based on this launch configuration the Java code generator constructs an entry point (a `main` method really) that serves as an entry point for the generated code. Launch configuration based code generation can be initiated using the `Ctrl+Alt+B` shortcut.

Upon completion of the code generation process the status is output to the console as shown in Figure 11.2. In particular this figure shows the status of code generating the `AlarmPP` model



Figure 11.1: Launching the Java code generator.

available in the Overture standard examples. As indicated by the console output, the generated code is available as an Eclipse project in the `<workspace>/<project>/generated/java` folder.

The Java code generator is also exposed as a Maven plugin in order to provide support for build and test automation. In essence this means that the Java code generator can be invoked from build environments that use Maven to manage Java projects. This allows code generated VDM models to be seamlessly integrated with other dependencies such as manually implemented components that the VDM model access via the Java bridge (see chapter 16), user-interfaces or third-party libraries. Another feature of this Maven plugin is that it allows model tests, written using `VDMUnit` to be code generated to JUnit4 tests, which can be executed via Maven in order to validate the generated code. An online tutorial that demonstrates how to use the Maven plugin is available via [DelegateTutorial].

11.2 Configuration of the Java Code Generator

The Java code generator can be configured via a preference page as shown in Figure 11.3. The preference page can be accessed in the way you would normally access an Eclipse preference page or via the context menu shown above in Figure 11.1. The Java code generator provides a few

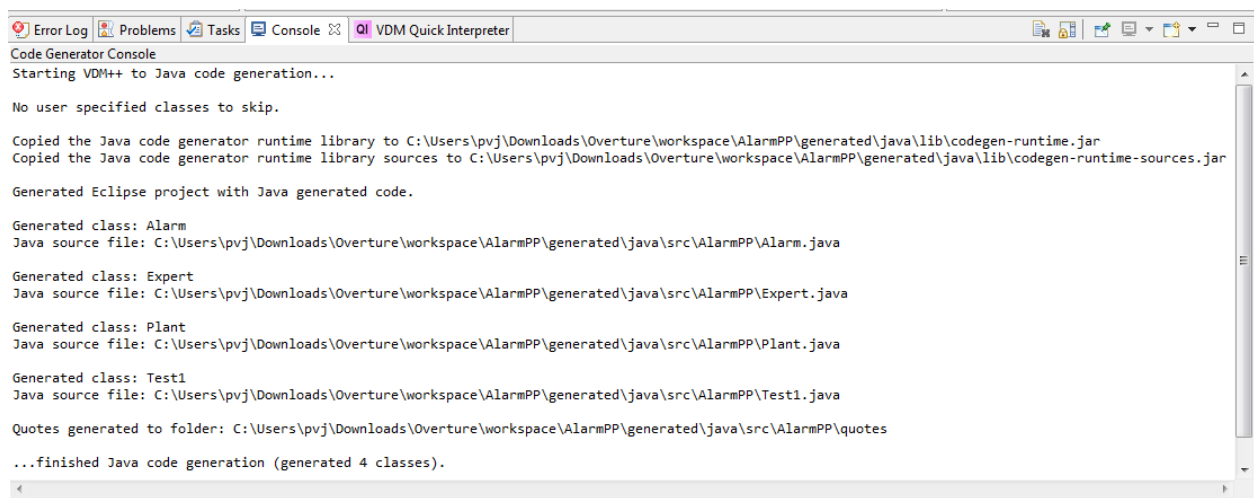


Figure 11.2: The status of code generating the AlarmPP example.

options that allows the user to configure the code generation process (see Figure 11.3). The subsections below treat each of these configuration parameters individually, in the order they appear in the preference page.

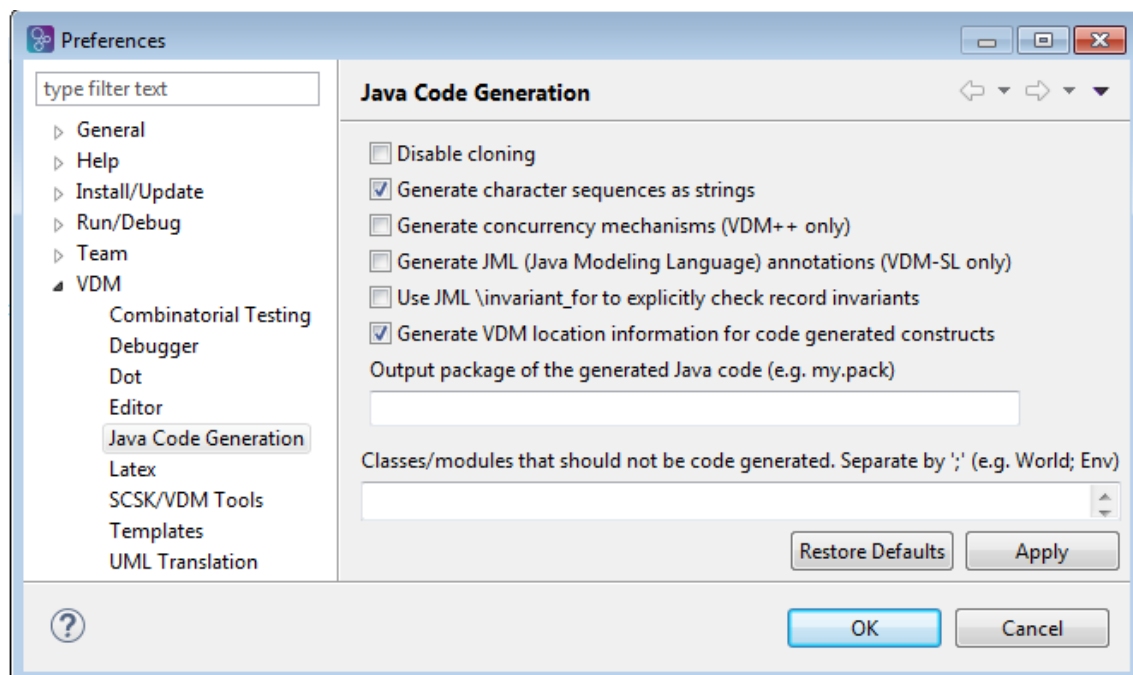


Figure 11.3: Configuration of the Java code generator.



11.2.1 Disable cloning

In order to respect the value semantics of VDM the Java code generator sometime needs to perform deep copying of objects that represent composite value types (records, tuples, tokens, sets, sequences and maps). For example, in VDM a record is a value type, which means that occurrences of the record must be copied when it appears in the right-hand side of an assignment, it is passed as an argument or returned as a result. However, Java does not support composite value types like structs and records, and as a consequence record types must be represented using classes, which use reference semantics. This means that an object reference, which is used to represent a composite value type in the generated Java code must be deep copied when it appears in the right-hand side of an assignment, it is passed as an argument or returned as a result. For arbitrarily complex value types (such as records composed of record or sets of composed of sets) deep copying may introduce a significant overhead in the generated code. If the specification subject to code generation does not truly rely on value semantics the user may wish to disable deep copying of value types in the generated code in order to remove this overhead. The user should, however, be aware that disabling of cloning may lead to code being generated that does not preserve the semantics of the input specification and in general disabling of cloning is discouraged. By default cloning is enabled.

11.2.2 Generate character sequences as strings

In VDM a string is a sequence of characters and there is no notion of a string type. Java in particular works differently since it uses a separate type to represent a string. The default behaviour of the Java code generator is to code generate sequences of characters as strings and subsequently do the necessary conversion between between strings and sequences in the generated code. Another possibility is to treat a string literal for what it truly is, namely a sequence of characters, and thereby avoid any conversion between strings and sequences. In order to do that, i.e. *not* generating character sequences as strings, the corresponding option must be unchecked.

11.2.3 Generate concurrency mechanisms

If the user does not rely on the concurrency mechanisms of VDM++ and does not want to include support for them in the generated code the corresponding option in the preference page must be unchecked. By default the behaviour of the Java code generator is to not include support for the concurrency mechanisms of VDM++ in the generated code.

11.2.4 Generate Java Modeling Language (JML) annotations

When a VDM model is code generated to Java all the contract-based elements of the model, i.e. the pre conditions, post conditions and invariants, are ignored by default. When this option is selected the contract-based elements of a VDM-SL model are translated to JML annotations [Burdy&05] that are added to the generated Java code. This allows the system properties, expressed in terms of



pre conditions, post conditions and invariants, to be checked against the generated code. The generated JML annotated programs can be checked for correctness using a JML tool such as OpenJML¹.

11.2.5 Use JML `\invariant_for` to explicitly check record invariants

The JML generator offers two ways to perform the invariant check of a record. By default the JML generator does this by invoking a `valid` method generated for each record definition. When this option is enabled the JML generator instead uses the JML `\invariant_for` construct to perform the record invariant checks. Note that the `\invariant_for` construct is currently not supported by OpenJML, which is the reason why this option is not enabled by default.

11.2.6 Generate VDM location information for code generated constructs

When a VDM model is code generated it can be helpful to know where the constructs in the generated code originate from. When this option is enabled the Java code generator will generate VDM location information for methods, statements and local declarations in the generated code. More specifically, the Java code generator will generate a Java source code comment containing the name of the VDM source file and the line number and the position, for each method, statement and local declaration. As an example, the code fragment below says that the Java return statement originates from a VDM construct at line 25, position 12 in `File.vdmsl`.

```
/* File.vdmsl 25:12 */  
return 42;
```

11.2.7 Choose output package

The Java code generator allows the output package of the generated code to be specified. If the user does not specify a package, the code generator outputs the generated Java code to a package with the same name as the VDM project. If the name of the project is not a valid Java package, then the generated code is output to the default Java package.

11.2.8 Skip classes/modules during the code generation process

It may not always make sense to code generate every class or module in a VDM project. A class or module can often be skipped if it acts as an execution entry point or it is used to load input for the specification. Classes or modules that the user wants to skip can be specified in the text box in the Java code generator preference page by separating the class/module names by a semicolon. As an example, `World; Env` makes the code generator skip code generation of `World` and `Env`, while generating code for any other module or class. For convenience the output of the Java code generator will also inform the user about what classes or modules are being skipped.

¹<http://www.openjml.org>



11.3 Limitations of the Java Code Generator

If the Java code generator encounters a construct that it cannot code generate it will report it as unsupported to the user and the user can then try to rewrite that part of the specification using other (supported) constructs. Reporting of unsupported constructs is done via the console output and using editor markers. In order to demonstrate this, Figure 11.4 shows the console output of the Java code generator when it encounters a type bind, which is an example of an unsupported language construct. Note the small marker appearing in the editor in order to point out where use of the construct appears. For the type bind example in Figure 11.4 the following message is reported:

Following VDM constructs are not supported by the code generator:
b:(bool * bool) (ATypeMultipleBind) at 6:14. Reason:
Type binds are not supported

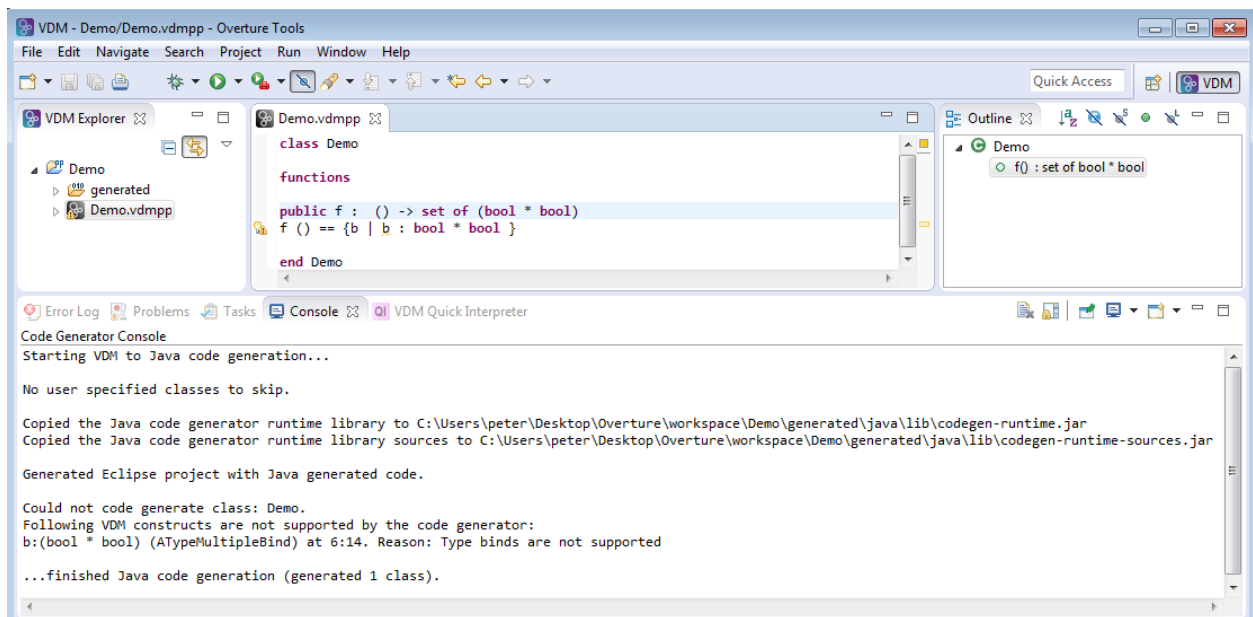


Figure 11.4: Reporting of unsupported constructs in the console.

The user will get similar messages and markers for other unsupported VDM constructs. To summarise, the Java code generator currently does not support code generation of multiple inheritance and neither does it support traces, type binds, invariant checks and pre and post conditions. Furthermore, let expressions appearing on the right-hand side of an assignment will also be reported as unsupported. The Java code generator also does not support every pattern. The patterns that are currently not supported are: object, map union, map, union, set, sequence, concatenation and match value.



11.4 The Code Generation Runtime Library

The generated code relies on a runtime library used to represent some of the types available in VDM (tokens, tuples etc.) as well as collections and support for some of the complex operators such as sequence modifications. For simplicity every Eclipse project generated by the Java code generator contains the runtime library. More specifically, there is a copy of the runtime library containing only the binaries (`lib/codegen-runtime.jar`) as well as a version of the runtime library that has the source code attached (`lib/codegen-runtime-sources.jar`). The runtime library is imported by every code generated class using the Java import statement `import org.overture.codegen.runtime.*;` and in order to compile the generated Java code the runtime library must be visible to the Java compiler.

Similar to VDMTools the runtime library also provides implementation for subset of the functionality available in the standard libraries: The runtime library provides a full implementation of the MATH library, support for conversion of values into character sequences as provided by the `VDMUtil`, and finally functionality to write to the console as available in the IO library.

11.5 Translation of the VDM types and type constructors

Table 11.1 describes how the VDM type(s) in the left column are represented in the generated Java code (the right column). In this table `pack` is the user-specified root package of the generated Java code (see section 11.2.7) and `E`, `D` and `R` represent arbitrary VDM types. The type mapping in the last row is only used when the *Generate character sequences as strings* option is selected (see section 11.2.2). Some of the types used to represent the VDM types are native Java types (from package `java.lang`), others are part of the Java code generator runtime library (from package `org.overture.codegen.runtime`), and some are generated.

VDM type(s)	Java type
bool	<code>java.lang.Boolean</code>
nat, nat1, int, rat, real	<code>java.lang.Number</code>
char	<code>java.lang.Character</code>
token	<code>org.overture.codegen.runtime.Token</code>
Tuple types (e.g. nat * nat)	<code>org.overture.codegen.runtime.Tuple</code>
Union types (e.g. nat nat)	<code>java.lang.Object</code>
Quote type <code><T></code>	<code>pack.quotes.TQuote</code>
User-defined types <code>T = D</code>	Represented using the representation of type <code>D</code>
A class <code>C</code>	<code>pack.C</code>
Record type <code>R</code> defined in class or module <code>M</code>	Inner class <code>pack.M.R</code>
set of E	<code>org.overture.codegen.runtime.VDMSet</code>
map D to R, inmap D to R	<code>org.overture.codegen.runtime.VDMMap</code>
seq of E, seq1 of E	<code>org.overture.codegen.runtime.VDMSeq</code>
seq of char, seq1 of char	<code>java.lang.String</code>



Table 11.1: The type mappings used by the Java code generator.

11.6 The C Code Generator

Translation of VDM models to C code is a new feature of Overture. The target language of the C generator is the executable subset of VDM-RT. It is not yet feature-complete, but it can handle models that do not use sophisticated features of VDM-RT.

In this section we give an overview of the current state of language feature support. The translation strategy is then described in some detail as a means of providing a tutorial introduction to calling into the generated model.

11.6.1 Language Feature Support Status

Overture's C code generator is under active development. At present, items in black are tested and known, to the best of the developers' knowledge, to translate correctly to C. Items in **red** are not yet supported.

- Classes.
- Inheritance.
- Overloading and overriding.
- Nested constructor calling.
- The `self` expression.
- Type definitions.
 - **Type invariants.**
- Boolean and numeric expressions.
 - **Quantifiers.**
- Let expressions.
- `is_(var, type)` testing for basic types.
- Records.
- **Products.**
- Sets, sequences and maps.
 - **Nested state designators, e.g. `a(i)(j) := k`.**



- Explicit functions and operations.
 - **Lambda abstractions.**
 - **Pre- and post-conditions.**
- Loops.
 - **For index loops.**
 - **While loops.**
- **Real-time and distribution features.**
 - The `time` statement.
- I/O library.
 - **String patterns.**
 - **File I/O.**
- MATH library.
 - **`rand()`**
 - **`srand()`**
 - **`srand2()`**

Like the Java code generator, the C code generator can be invoked from the context menu in the VDM project explorer, as shown in in Figure 11.5. Unlike the Java code generator, it must be

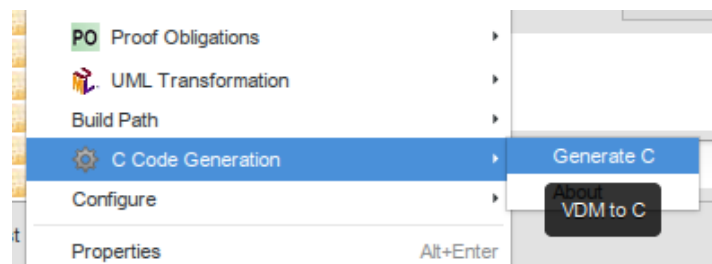


Figure 11.5: Invoking the C code generator.

installed through Overture’s “Install New Software ...” feature. The repository URL is

<http://overture.au.dk/vdm2c/master/repository/>.



Code is emitted to the folder `generated/src/c` under the corresponding VDM project. Please note that this folder is wiped clean each time code is generated. Additional development work on the generated code *must not* be carried out in this folder.

The generator will emit a `main.c` file containing an empty `main()` function. The generator also emits a CMake file called `ProjectCMakeLists.txt`. When renamed to `CMakeLists.txt`, executing

```
cmake .
make
```

in the generated code root directory will compile an executable, but inert, binary of the generated model. At this point the user is free to populate the `main` function. The header file `MangledNames.h` contains the correspondences between model identifiers and the mangled names used internally in the generated code. This file can be used to call into the generated model using approximately the naming convention of the VDM model.

11.6.2 The C Translation Strategy

The priority of the translation strategy is to remain faithful to the VDM-RT semantics described above. The strategy therefore assumes that VDM-RT specifications have been validated using Overture's various facilities. This section describes the strategy and the two sides of the translation mechanism, the implementation of the strategy and the native C support library. The generated code can be compiled and executed by means of a CMake build mechanism and an empty `main.c` file, but no calls into the generated code proper are made.

Native Support Library

Implementations generated from VDM-RT models consist of two parts, the generated code and a native support library². The native library is fixed and does not change during the code generation process. We illustrate its design here by means of very simple generated VDM models.

The native library provides a single fundamental data structure in support of all the VDM-RT data types, called `TypedValue`. The complete definition is shown in Listing 11.1. A pointer to `TypedValue` is #defined as `TVP`, and is used throughout the implementation.

Listing 11.1: Fundamental code generator data type.

```
typedef enum {
    VDM_INT, VDM_NAT, VDM_NAT1, VDM_BOOL, VDM_REAL,
```

²The design of the native library is based on the following four sources:

http://www.pvv.ntnu.no/~h_akonhal/main.cgi/c/classes/, accessed 2016-09-22.

<http://www.eventhelix.com/RealtimeMantra/basics/ComparingCPPAndCPerformance2.htm>, accessed 2016-09-22.

<http://www.go4expert.com/articles/virtual-table-vptr-multiple-inheritance-t16616/>, accessed 2016-09-22.

<http://www.go4expert.com/articles/virtual-table-vptr-t16544/>, accessed 2016-09-22.



```
VDM_RAT, VDM_CHAR, VDM_SET, VDM_SEQ, VDM_MAP,
VDM_PRODUCT, VDM_QUOTE, VDM_RECORD, VDM_CLASS
} vdmtype;

typedef union TypedValueType {
    void* ptr;                // VDM_SET, VDM_SEQ, VDM_CLASS,
                             // VDM_MAP, VDM_PRODUCT
    int intVal;               // VDM_INT and INT1
    bool boolVal;            // VDM_BOOL
    double doubleVal;        // VDM_REAL
    char charVal;            // VDM_CHAR
    unsigned int uintVal;    // VDM_QUOTE
} TypedValueType;

struct TypedValue {
    vdmtype type;
    TypedValueType value;
};

struct Collection {
    struct TypedValue** value;
    int size;
};
```

An element of this type carries information about the type of the VDM value represented and the value proper. For space efficiency, the value storage mechanism is a C union.

Members of the basic, unstructured types `int`, `char`, *etc.* are stored directly as values in corresponding fields. Due to subtype relationships between certain VDM types, for instance `nat` and `nat1`, fields in the union can be reused. Functions to construct such basic values are provided:

- `TVP newInt(int)`
- `TVP newBool(bool)`
- `TVP newQuote(unsigned int)`
- *etc.*

All the operations defined by the VDM language manual on basic types are implemented one-to-one. They can be found in the header file

`VdmBasicTypes.h`.

Members of structured VDM types, such as `seq` and `set`, are stored as references, owing to their variable size. The `ptr` field is dedicated to these. These collections are represented as arrays of `TypedValue` elements, wrapped in the C structure `Collection`. The field `size` of `Collection` records the number of elements in the collection. Naturally, collections can be nested. At the level of VDM these data types are immutable and follow value semantics. But internally they are constructed in various ways. For instance, internally creating a fresh set from known values is different from constructing one value by value according to some filter on values. In the



former case a new set is created in one shot, whereas in the latter an empty set is created to which values are added. Several functions are provided for constructing collections which accommodate these different situations.

- `newSetVar(size_t, ...)`
- `newSetWithValues(size_t, TVP*)`
- `newSeqWithValues(size_t, TVP*)`
- *etc.*

These rely on two functions for constructing the inner collections of type `struct Collection` at field `ptr`:

- `TVP newCollection(size_t, vdmtype)`
- `TVP newCollectionWithValues(size_t, vdmtype, TVP*)`

The former creates an empty collection that can be grown as needed by memory re-allocation. The latter wraps an array of values for inclusion in a TVP value of structured type. All the operations defined in the VDM language manual on structured types are implemented one-to-one. They can be found in the header files `VdmSet.h`, `VdmSeq.h` and `VdmMap.h`.

VDM's object orientation features are fundamentally implemented in the native library using C `structs`. In brief, a class is represented by a `struct` whose fields represent the fields of the class. The functions and operations of the class are implemented as functions associated with the corresponding `struct`.

Consider the following example VDM specification.

Listing 11.2: Example VDM model.

```
class A
instance variables
    private i : int := 1;

operations
    public op : () ==> int
    op() ==
        return i;
end A
```

The code generator produces the two files `A.h` and `A.c`, shown below.

Listing 11.3: Corresponding header file `A.h`.

```
#include "Vdm.h"
#include "A.h"

#define CLASS_ID_A_ID 0
```




```
#define ACLASS struct A*

#define CLASS_A__Z2opEV 0

struct A
{
    VDM_CLASS_BASE_DEFINITIONS(A);

    VDM_CLASS_FIELD_DEFINITION(A,i);
};

TVP __Z1AEV(AClass this_);

AClass A_Constructor(AClass);
```

The basic construct is a struct containing the class fields and the class virtual function table:

Listing 11.4: Macro for defining class virtual function tables.

```
#define VDM_CLASS_FIELD_DEFINITION(className, name) \
    TVP m_##className##_##name

#define VDM_CLASS_BASE_DEFINITIONS(className) \
    struct VTable * __##className##_pVTable; \
    int __##className##_id; \
    unsigned int __##className##_refs
```

The virtual function table contains information necessary for resolving a function call in a multiple inheritance context as well as a field which receives a pointer to the implementation of the operation op.

Listing 11.5: Virtual function table.

```
typedef TVP (*VirtualFunctionPointer)(void * self, ...);

struct VTable
{
    //Fields used in the case of multiple inheritance.
    int d;
    int i;

    VirtualFunctionPointer pFunc;
};
```

The rest of the important parts of the implementation consists of the function implementing op(), the definition of the virtual function table into which this slots and the complete constructor mechanism.

Listing 11.6: Corresponding implementation file A.c.

```
#include "A.h"
```



```
#include <stdio.h>
#include <string.h>

void A_free_fields(struct A *this)
{
    vdmFree(this->m_A_i);
}

static void A_free(struct A *this)
{
    --this->_A_refs;
    if (this->_A_refs < 1)
    {
        A_free_fields(this);
        free(this);
    }
}

/* A.vdmrt 6:9 */
static TVP _Z2opEV(AClass this)
{
    /* A.vdmrt 8:10 */
    TVP ret_1 = vdmClone(newBool(true));

    /* A.vdmrt 8:3 */
    return ret_1;
}

static struct VTable VTableArrayForA [] =
{
    {0,0,((VirtualFunctionPointer) _Z2opEV),},
};

AClass A_Constructor(AClass this_ptr)
{
    if(this_ptr==NULL)
    {
        this_ptr = (AClass) malloc(sizeof(struct A));
    }

    if(this_ptr!=NULL)
    {
        this_ptr->_A_id = CLASS_ID_A_ID;
        this_ptr->_A_refs = 0;
        this_ptr->_A_pVTable=VTableArrayForA;

        this_ptr->m_A_i= NULL ;
    }
}
```



```

        return this_ptr;
    }

    // Method for creating new "class"
    static TVP new()
    {
        ACLASS ptr=A_Constructor(NULL);

        return newTypeValue(VDM_CLASS,
                           (TypedValueType)
                           { .ptr=newClassValue(ptr->_A_id,
                                                &ptr->_A_refs,
                                                (freeVdmClassFunction) &A_free,
                                                ptr) });
    }

    /* A.vdmrt 1:7 */
    TVP _Z1AEV(ACLASS this)
    {
        TVP __buf = NULL;

        if(this == NULL)
        {
            __buf = new();

            this = TO_CLASS_PTR(__buf, A);
        }

        return __buf;
    }

```

TO_CLASS_PTR merely unwraps values and can be ignored for now.

Construction of an instance of class A starts with a call to `_Z1AEV`. An instance of `struct A` is allocated and its virtual function table is populated with the pointer to the implementation of `op()`, `_Z2opEV`. The latter name is a result of a name mangling scheme implemented in order to avoid name clashes in the presence of inheritance³. A header file called `MangledNames.h` provides the mappings between VDM model identifiers and mangled names in the generated code. This mapping aids in writing the `main` function. The scheme used is `ClassName_identifier`. Listing 11.7 shows the contents of the file for the example model.

Listing 11.7: File `MangledNames.h`.

```

#define A_op _Z2opEV
#define A_A _Z1AEV

```

³The name mangling scheme is based on the following sources:

https://en.wikipedia.org/wiki/Name_mangling, accessed 2016-09-28.

<http://www.avabodh.com/cxxin/namemangling.html>, accessed 2016-09-28.



By default, the code generation process provides an empty `main.c` file such that it is possible to compile the generated code initially. It will, of course, be completely inert. The following example populated `main.c` file illustrates how to make use of the generated code.

Listing 11.8: Example `main.c` file.

```
#include "A.h"

int main()
{
    TVP a_instance = _Z1AEV(NULL);
    TVP result;

    result = CALL_FUNC(A, A, a_instance, CLASS_A__Z2opEV);

    printf("Operation op returns:  %d\n", result->value.intVal);

    vdmFree(result);
    vdmFree(a_instance);

    return 0;
}
```

Had the class `A` contained any values or static fields, the very first calls into the model would have been to `A_const_init()` and `A_static_init()`. As this is not the case here, an instance of the class implementation is first created, together with a variable to store the result of `op`. The macro `CALL_FUNC` carries out the necessary calculations for calling the correct version of `_Z2opEV` in the presence of inheritance and overriding (not the case here).

Listing 11.9: Macros supporting function calls.

```
#define GET_STRUCT_FIELD(tname,ptr,fieldtype,fieldname) \
    (*((fieldtype*)((unsigned char*)ptr) + \
        offsetof(struct tname,fieldname)))

#define GET_VTABLE_FUNC(thisTypeName,funcTname,ptr,id) \
    GET_STRUCT_FIELD(thisTypeName,ptr,struct VTable*, \
        _##funcTname##_pVTable)[id].pFunc

#define CALL_FUNC(thisTypeName,funcTname,classValue,id, args... ) \
    GET_VTABLE_FUNC( thisTypeName, \
        funcTname, \
        TO_CLASS_PTR(classValue,thisTypeName), \
        id) \
    (CLASS_CAST(TO_CLASS_PTR(classValue,thisTypeName), \
        thisTypeName, \
        funcTname), ## args)
```

The result is assigned to `result`, which is then accessed according to the structure of `TVP`. The



function `vdmFree` is the main memory cleanup function for variables of type TVP.

Translating Features of VDM-SL

In this section we discuss how the basic features of VDM-RT, those contained in the subset VDM-SL, are translated to C.

Basic data types Instances of the fundamental data types of VDM-SL (integers, reals, characters *etc.*) translate directly to instances of type TVP with the appropriate field of the union structure `TypedValueType` set to the value of the instance. They are instantiated using the corresponding constructor functions `newInt()`, `newBool()` *etc.* introduced above. Operations on fundamental data types preserve value semantics by always allocating new memory for the result TVP instance and returning the corresponding pointer.

Structured types. Like basic types, aggregate types such as sets and maps are treated in exactly the same way. The support library provides both the data type infrastructure as well as the operations on aggregate types such that translation is rendered straightforward. For example, the expression

```
a : set of int := {1} union {2}
```

translates directly to

```
TVP a = vdmSetUnion(newSetVar(1, newInt(1)), newSetVar(1, newInt(2)));
```

where `newSetVar()` is one of the several special-purpose internal constructors. The translation strategy is similar for sequences and maps. Value semantics for these immutable data types is maintained in the same way as for the basic data types.

Quote types Quote types such as that shown in Listing 11.10 are treated at the individual element level. Each element is assigned a unique number via a `#define` directive, as shown in Listing 11.10.

Listing 11.10: Quote type example.

```
class QuoteExample
types
    public QuoteType = <Val1> | <Val2> | <Val3>
end QuoteExample
```

Listing 11.11: Quote type example translation.

```
...
```



```
#ifndef QUOTE_VAL1
#define QUOTE_VAL1 2658640
#endif /* QUOTE_VAL1 */

#ifndef QUOTE_VAL2
#define QUOTE_VAL2 2658641
#endif /* QUOTE_VAL2 */

...
```

Union types. The decision to keep run-time type information for every variable of type TVP obviates the need for a translation strategy for union types.

Translating Features of VDM++

Classes Earlier we introduced the mechanism of C structures used to represent classes. Translation of a model class is therefore straightforward, with each class receiving its own specific `struct`. As illustrated in Listings 11.3 and 11.6 above, each class receives its own pair of C header and implementation files. Most importantly, the header file contains the definition of the corresponding class `struct` and the declarations of the interface functions for this `struct`. These include the top-level constructor and initialization and cleanup functions for class `values` and `static` field declarations. The implementation (`.c`) file contains the constructor mechanism, the definition of the virtual function table of the class and the implementations of the class's functions and operations. The virtual function table is constructed in accordance with the inheritance hierarchy in which the class belongs (this is discussed below). Class `values` definitions and static fields are implemented as global variables. Their definitions are also inserted in the implementation file, along with initializer and cleanup functions to be called, respectively, when the implementation starts and terminates.

Inheritance The effect of inheritance is to augment the definition of the inheriting class with the features of the parent class, *modulo* overriding. In our `struct`-based implementation of classes and objects, the traits of the base class are copied into the `struct` corresponding to the inheriting class. Therefore, the `struct` of the inheriting class duplicates the fields and virtual function table of the base class.

Consider the translation of the following model with inheritance. Despite its cumbersome length, we provide the listing of the complete translation so that the reader may also gain familiarity (at his/her own pace) with all the elements of the generated code.

```
class A
instance variables
public field_A : int := 0;

operations
public opA : int ==> int
```



```

opA(i) == return i;
end A

class B is subclass of A
operations
public opB : () ==> ()
opB() == skip;
end B

class C
instance variables
b : B := new B();

operations
public op : () ==> int
op() == return b.opA(b.field_A);
end C

```

The six files A.h, A.c, B.h, B.c, C.h and C.c reproduced below make up the complete translation.

Listing 11.12: "File A.h."

```

// The template for class header
#ifndef CLASSES_A_H_
#define CLASSES_A_H_

#define VDM_CG

#include "Vdm.h"

//include types used in the class
#include "A.h"

extern TVP numFields_1;

#define CLASS_ID_A_ID 0
#define ACLASS struct A*

#define CLASS_A__Z3opAEI 0

struct A
{
    VDM_CLASS_BASE_DEFINITIONS(A);

    VDM_CLASS_FIELD_DEFINITION(A, field_A);
    VDM_CLASS_FIELD_DEFINITION(A, numFields);
};

```



```
TVP _Z1AEV(AClass this_);

void A_const_init();
void A_const_shutdown();
void A_static_init();
void A_static_shutdown();

void A_free_fields(AClass);
AClass A_Constructor(AClass);

#endif /* CLASSES_A_H */
```

Listing 11.13: "File A.c."

```
#include "A.h"
#include <stdio.h>
#include <string.h>

void A_free_fields(struct A *this)
{
    vdmFree(this->m_A_field_A);
}

static void A_free(struct A *this)
{
    --this->_A_refs;
    if (this->_A_refs < 1)
    {
        A_free_fields(this);
        free(this);
    }
}

static TVP _Z17fieldInitializer2EV(){
    TVP ret_1 = vdmClone(newInt(0));

    return ret_1;
}

static TVP _Z17fieldInitializer1EV(){
    TVP ret_2 = vdmClone(newInt(1));

    return ret_2;
}

static TVP _Z3opAEI(AClass this, TVP i){
    TVP ret_3 = vdmClone(i);

    return ret_3;
}
```




```

}

void A_const_init(){
    numFields_1 = _Z17fieldInitializer1EV();

    return ;
}

void A_const_shutdown(){
    vdmFree(numFields_1);

    return ;
}

void A_static_init(){

    return ;
}

void A_static_shutdown(){

    return ;
}

static struct VTable VTableArrayForA  [] ={

    {0,0,((VirtualFunctionPointer) _Z3opAEI),},
} ;

AClass A_Constructor(AClass this_ptr)
{
    if(this_ptr==NULL)
    {
        this_ptr = (AClass) malloc(sizeof(struct A));
    }

    if(this_ptr!=NULL)
    {
        this_ptr->_A_id = CLASS_ID_A_ID;
        this_ptr->_A_refs = 0;
        this_ptr->_A_pVTable=VTableArrayForA;

        this_ptr->m_A_field_A= _Z17fieldInitializer2EV();
    }

    return this_ptr;
}

static TVP new(){
    AClass ptr=A_Constructor(NULL);

```



```
        return newTypeValue(VDM_CLASS, (TypedValueType)
                        { .ptr=newClassValue(ptr->_A_id, &ptr->_A_refs,\
                        (freeVdmClassFunction)&A_free, ptr)});
    }

TVP _Z1AEV(AClass this){
    TVP __buf = NULL;

    if ( this == NULL )
    {
        __buf = new();

        this = TO_CLASS_PTR(__buf, A);
    }

    return __buf;
}

TVP numFields_1 =  NULL ;
```

Listing 11.14: "File B.h."

```
#ifndef CLASSES_B_H_
#define CLASSES_B_H_

#define VDM_CG

#include "Vdm.h"
#include "A.h"
#include "B.h"

#define CLASS_ID_B_ID 1

#define BCLASS struct B*

#define CLASS_B__Z3opBEV 0

struct B
{
    VDM_CLASS_BASE_DEFINITIONS(A);

    VDM_CLASS_FIELD_DEFINITION(A, field_A);
    VDM_CLASS_FIELD_DEFINITION(A, numFields);

    VDM_CLASS_BASE_DEFINITIONS(B);

    VDM_CLASS_FIELD_DEFINITION(B, numFields);
};

TVP _Z1BEV(BCLASS this_);
```



```
void B_const_init();
void B_const_shutdown();
void B_static_init();
void B_static_shutdown();

void B_free_fields(BCLASS);
BCLASS B_Constructor(BCLASS);

#endif /* CLASSES_B_H_ */
```

Listing 11.15: "File B.c."

```
#include "B.h"
#include <stdio.h>
#include <string.h>

void B_free_fields(struct B *this)
{
}

static void B_free(struct B *this)
{
    --this->_B_refs;
    if (this->_B_refs < 1)
    {
        B_free_fields(this);
        free(this);
    }
}

static void _Z3opBEV(BCLASS this){
{
    //Skip
};
}

void B_const_init(){
    return ;
}

void B_const_shutdown(){
    return ;
}

void B_static_init(){
    return ;
}

void B_static_shutdown(){
    return ;
}
```



```
static struct VTable VTableArrayForB [] ={

    {0,0,((VirtualFunctionPointer) _Z3opBEV),},

} ;

BCLASS B_Constructor(BCLASS this_ptr)
{
    if(this_ptr==NULL)
    {
        this_ptr = (BCLASS) malloc(sizeof(struct B));
    }

    if(this_ptr!=NULL)
    {
        A_Constructor((AClass) CLASS_CAST(this_ptr,B,A));

        this_ptr->_B_id = CLASS_ID_B_ID;
        this_ptr->_B_refs = 0;
        this_ptr->_B_pVTable=VTableArrayForB;
    }

    return this_ptr;
}

static TVP new(){
    BCLASS ptr=B_Constructor(NULL);

    return newTypeValue(VDM_CLASS, (TypedValueType)
        {
            .ptr=newClassValue(ptr->_B_id, &ptr->_B_refs,\
                (freeVdmClassFunction)&B_free, ptr));
}

TVP _Z1BEV(BCLASS this){
    TVP __buf = NULL;

    if ( this == NULL )
    {
        __buf = new();
        this = TO_CLASS_PTR(__buf, B);
    }

    _Z1AEV(((AClass) CLASS_CAST(this, B, A)));

    return __buf;
}
```

Listing 11.16: "File C.h."

```
#ifndef CLASSES_C_H_
#define CLASSES_C_H_
```



```

#define VDM_CG

#include "Vdm.h"
#include "B.h"
#include "C.h"

extern TVP numFields_2;

#define CLASS_ID_C_ID 2

#define CCLASS struct C*

#define CLASS_C__Z2opEV 0

struct C
{
    VDM_CLASS_BASE_DEFINITIONS(C);

    VDM_CLASS_FIELD_DEFINITION(C,b);
    VDM_CLASS_FIELD_DEFINITION(C,numFields);
};

TVP _Z1CEV(CCLASS this_);

void C_const_init();
void C_const_shutdown();
void C_static_init();
void C_static_shutdown();

void C_free_fields(CCLASS);
CCLASS C_Constructor(CCLASS);

#endif /* CLASSES_C_H_ */

```

Listing 11.17: "File C.c."

```

#include "C.h"
#include <stdio.h>
#include <string.h>

void C_free_fields(struct C *this)
{
    vdmFree(this->m_C_b);
}

static void C_free(struct C *this)
{
    --this->_C_refs;
    if (this->_C_refs < 1)
    {

```



```
        C_free_fields(this);
        free(this);
    }
}

static TVP _Z17fieldInitializer4EV() {
    TVP ret_4 = vdmClone(_Z1BEV(NULL));

    return ret_4;
}

static TVP _Z17fieldInitializer3EV() {
    TVP ret_5 = vdmClone(newInt(1));

    return ret_5;
}

static TVP _Z2opEV(CCLASS this) {
    TVP embedding_1 = \
        GET_FIELD(A, A, GET_FIELD_PTR(C, C, this, b), field_A);

    TVP ret_6 = vdmClone(CALL_FUNC(B, A, GET_FIELD_PTR(C, C, this, b), \
        CLASS_A__Z3opAEI, embedding_1));

    return ret_6;
}

void C_const_init() {
    numFields_2 = _Z17fieldInitializer3EV();

    return ;
}

void C_const_shutdown() {
    vdmFree(numFields_2);

    return ;
}

void C_static_init() {
    return ;
}

void C_static_shutdown() {
    return ;
}

static struct VTable VTableArrayForC [] = {

    {0, 0, ((VirtualFunctionPointer) _Z2opEV), },

} ;
```



```

CCLASS C_Constructor(CCLASS this_ptr)
{
    if(this_ptr==NULL)
    {
        this_ptr = (CCLASS) malloc(sizeof(struct C));
    }

    if(this_ptr!=NULL)
    {
        this_ptr->_C_id = CLASS_ID_C_ID;
        this_ptr->_C_refs = 0;
        this_ptr->_C_pVTable=VTableArrayForC;

        this_ptr->m_C_b= _Zl7fieldInitializer4EV();
    }

    return this_ptr;
}

static TVP new(){
    CCLASS ptr=C_Constructor(NULL);

    return newTypeValue(VDM_CLASS, (TypedValueType)
        {
            .ptr=newClassValue(ptr->_C_id, &ptr->_C_refs,\
                (freeVdmClassFunction)&C_free, ptr));
}

TVP _Zl1CEV(CCLASS this){
    TVP __buf = NULL;

    if ( this == NULL )
    {
        __buf = new();

        this = TO_CLASS_PTR(__buf, C);
    }

    return __buf;
}

TVP numFields_2 = NULL ;

```

The duplication of the elements of A can be seen in the definition of struct B in B.h. The listing for C.c illustrates the mechanism by which a call to an inherited operation on an instance of B is achieved. The macro CALL_FUNC is the primary function and method call mechanism. It uses information about the type of the object on which the operation is invoked, as well as the class in which the operation is actually defined, to calculate a function pointer offset in the correct (duplicated) virtual function table of the instance of B. The class in which the operation is originally



defined (A in this case) is calculated by scanning the chain of superclasses of B and choosing the nearest definition. This method satisfies semantics of calls of inherited operations.

Polymorphism Overture limits polymorphism, the overloading of operations and functions. Overloaded operations and functions can only be distinguished by Overture's type system only if they differ in their parameter types. Operations differing only in return type can not be distinguished, rendering the following example definition illegal.

```
class Overloading

operations

public op : bool ==> ()
op(a) == skip;

public op : bool ==> bool
op(a) == return true;

end Overloading
```

Polymorphism is implemented by way of a name mangling scheme, whereby the name generated for any operation or function is augmented with tags representing its parameter types. For instance, the name of the following operation

```
public theOperation : int * bool * char ==> real
```

is generated as `_Z12theOperationEIBC`. The mangled name can be decomposed as follows:

- `_Z`: prepended to all mangled names.
- `12`: number of characters in the original name.
- `theOperation`: the original name.
- `E`: separator between name and parameter type tags.
- `I`: `int` parameter.
- `B`: `bool` parameter.
- `C`: `char` parameter.

Function and operation overriding In single inheritance scenarios, operation/function overriding is achieved in a simple way by choosing the overriding implementation closest in the inheritance chain to the class to which the object on which the operation is invoked belongs. This is in accordance with the corresponding semantics. In multiple inheritance scenarios, Overture does not allow ambiguity leading to a choice of implementation. For instance, the following model is illegal in Overture:



```
class A
operations
public op : () ==> bool
op () ==
    return true
end A

class B
operations
public op : () ==> bool
op () ==
    return false
end B

class C is subclass of B, A
end C
```

This forces the model developer to eliminate all such ambiguity, reducing the scenario that that of single inheritance.



Chapter 12

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. 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. The VDM model is typically used to define the implementation and constraints for each class and is therefore used for detailed semantic analysis. Note that state charts, activity diagrams, sequence diagrams, objects charts, package charts are not used in the UML mapping. It is essentially only the information used statically inside classes and their usage in class diagrams that is used.

To convert a UML class diagram model to a VDM++ model, you first need to export the UML model from Modelio to the Eclipse XMI format, called UML using the EMF UML3.0.0 format. At the moment, Modelio is the only UML tool supported. In particular, the exported UML models have been tested using Modelio 3.3.1. Export from Modelio is done as illustrated in Figure 12.1.

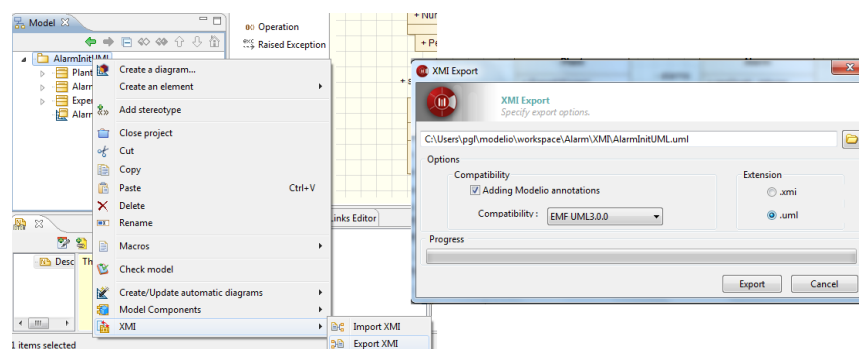


Figure 12.1: Exporting UML definitions from Modelio

Importing and exporting a UML model is an option in the Overture *Explorer* view.

Exporting: Right-click a VDM++ or VDM-RT project to access a submenu for *UML Transformation*. From here it is possible to *Convert to UML*. The resulting .uml file will be saved to the generated folder of your project.



Importing: To perform a UML import you must have the `.uml` file in the relevant project folder. You can either copy it manually or use the Eclipse *Import - File System* feature. Afterwards, it is possible to right-click the `.uml` file and choose the submenu for *UML Transformation* and then select *Convert to VDM*.

The mapping rules between VDM++/VDM-RT and UML models are further explained in Appendix H. Note that the mapping depends on the settings selected in the *Preferences* menu (see Figure 4.7 above). The general explanation of these options are:

Prefer associations during translation: If this option is chosen (ticked) references to instances of other classes will be modelled as associations between the classes in a class diagram. Otherwise these will be modelled as attributes.

Disable nested artifacts in Deployment Diagrams: If this option is chosen (ticked) nested artifacts in a deployment diagram are disabled; otherwise it is enabled.

Chapter 13

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 and it will initially be given the same name as the one generated from with postfix with “_VDM_RT”. 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. Also keep in mind that you may have some errors if your VDM++ model uses VDM-RT keywords as names.



Chapter 14

Graphics Plotting for VDM Models

On some occasions it is convenient to monitor how a VDM++ model’s variables evolve over time. To support this kind of analysis, a plugin extension to Overture enables graphics plotting of VDM models, as they are being interpreted. Currently plotting is only supported for instance variables that contain numeric values, and which have been declared **public** and **static**. In order to use this feature, it is necessary to install the “Overture Graphics Plugin” through Overture’s “Install New Software ...” menu. When the plugin has been installed it is possible to launch it as a special debug configuration as shown in Figure 14.1.

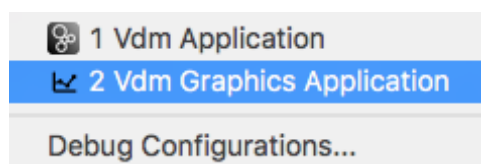


Figure 14.1: Starting up the debugger using the Graphics plugin

When the plugin is launched two files named `overture.graphics.properties` and `overture.graphics.jar` will be added to the project’s `lib` folder, and a new window will appear as shown in Figure 14.2. Next, the user needs to select the entry point to execute (a class and an operation). Note that both the class constructor and operation that constitute the entry point must not receive any arguments.

Afterwards, the user can define different plots based on a selection of instance variables – see Figure 14.3. Furthermore, from the file menu it is possible to save the plot configurations for later use. Numerous plots can be defined, if desired, although it is common to use the same plot to visualise multiple graphs.

Once a plot has been defined it can be selected via the “PLOTS” menu, and visualised by pressing the “Start” button as shown on the left of Figure 14.4. This will execute the user-defined entry point and plot the instance variables as the model execution advances over time. An example of one such plot is shown in Figure 14.4.

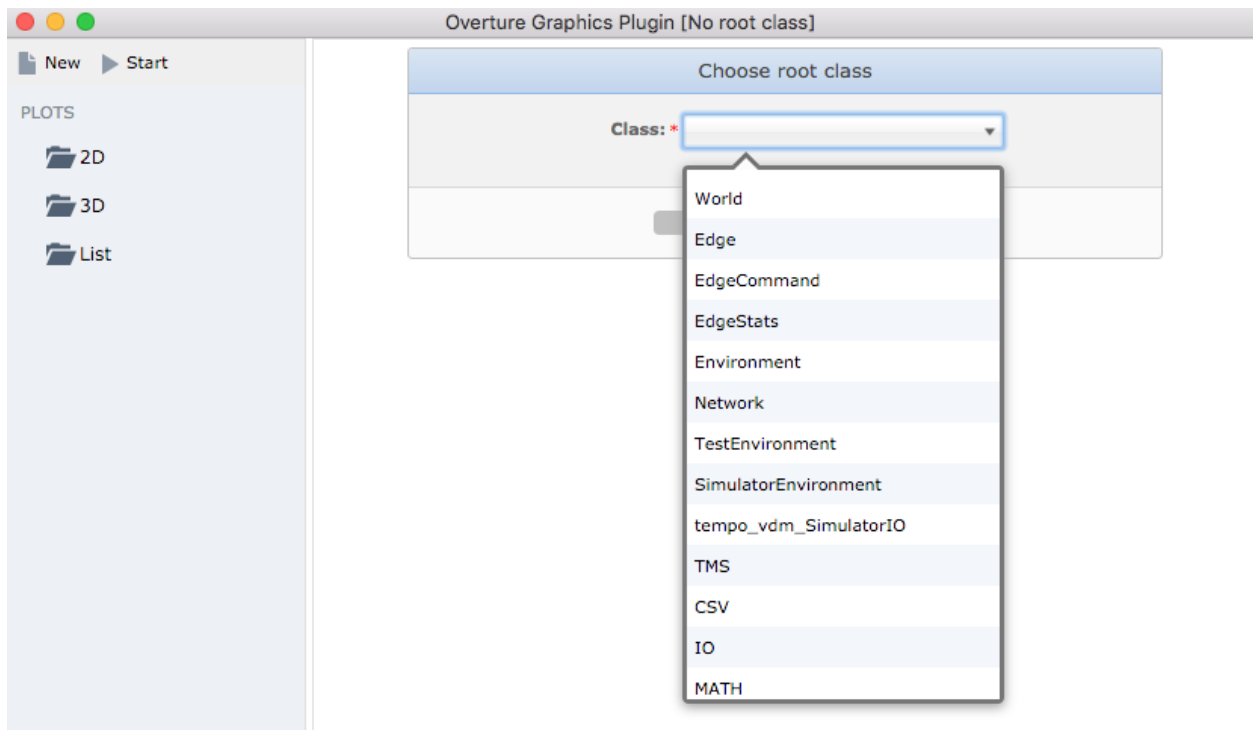


Figure 14.2: Starting up the debugger using the Graphics plugin

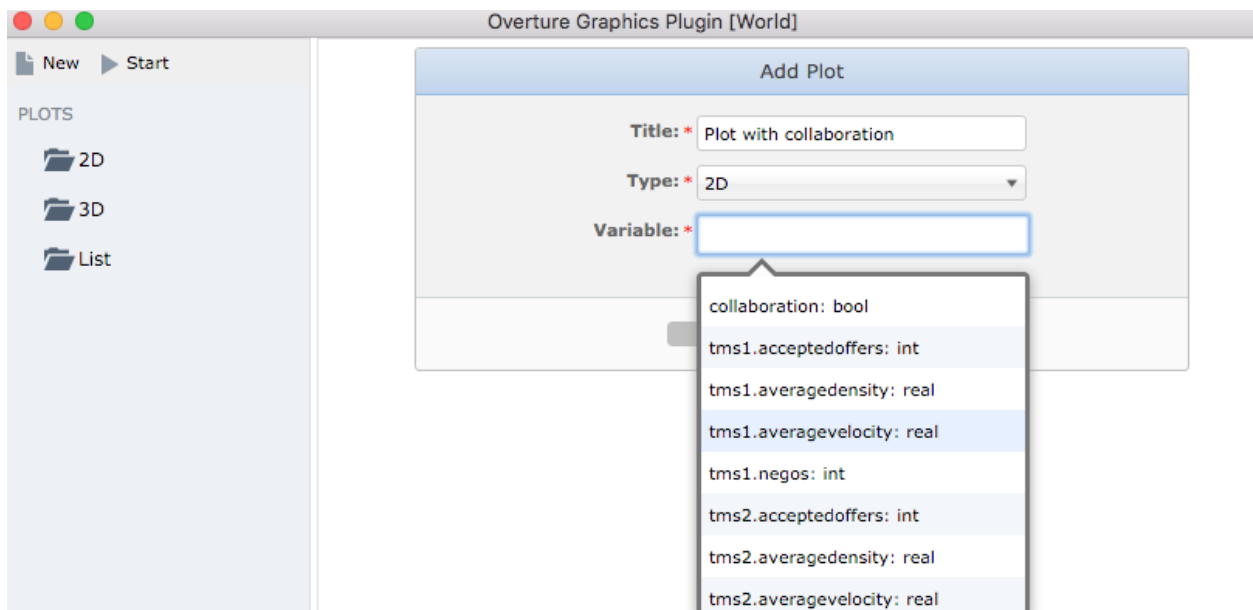


Figure 14.3: Starting up the debugger using the Graphics plugin

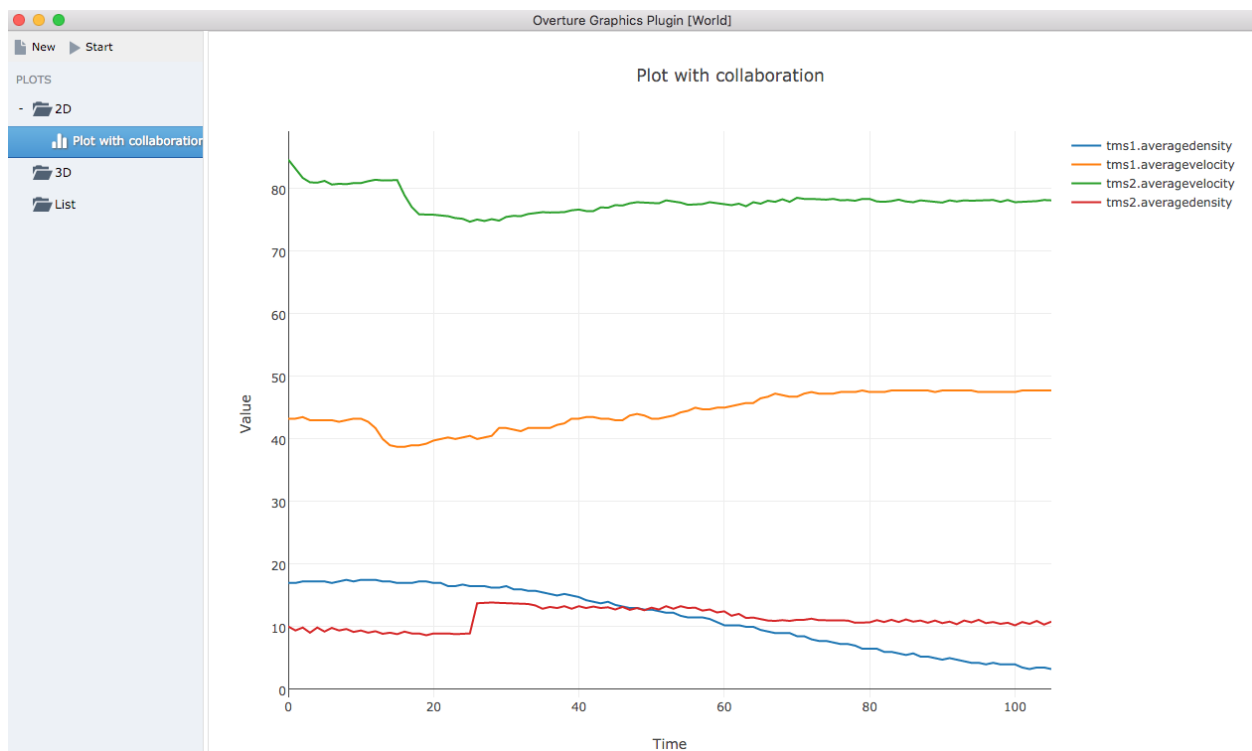


Figure 14.4: Starting the debugger using the Graphics plugin



Chapter 15

Analysing Logs from VDM-RT Executions



Whenever a VDM-RT model is executed a binary logfile with `.rtbin` extension is created in the `generated/logs` subfolder. In order to distinguish between multiple runs, the name of the logfile indicates the time at which the model was executed. Logfiles can be viewed with the built-in *RealTime Log Viewer*, by double-clicking the `.rtbin` file in the Explorer view. In addition, Overture provides a textual version of the log file with `.rt` extension, which can be inspected using any text editor.

The RealTime Log Viewer enables you to explore the simulated system execution in various ways. In Figure 15.1 the architectural overview of a system is shown, describing the CPU and BUS topology of the model.



Figure 15.1: Architectural overview

The RealTime Log Viewer also enables you to get an overview of the model execution at a system level – see Figure 15.2. 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  button or moving to the next time index using the **Move next** button. Also, Overture enables you to move to the previous time index using the **Move Previous** button. Finally, it is also possible to export all the generated views to JPEG formatted files using the  button. All the generated images will be placed in the logfile directory, where each image holds a name indicating the time of execution and the specific view (Execution overview, CPU1, CPU2 etc.).

The RealTime Log Viewer also provides an overview of all executions on a single CPU. Each CPU shows a detailed description of all operations and functions invoked on the particular CPU as



Figure 15.2: Execution overview

well as the scheduling of concurrent processes — see Figure 15.3.



Figure 15.3: Execution on a single CPU

Analyzing timing properties at the system level

The work of [Fitzgerald&07] presents an extension to VDM-RT, enabling the validation of system-level timing properties. *Validation conjectures* are descriptions of temporal relations between system level events, which can be evaluated over the execution trace. Later this work has been extended in [Ribeiro&11] to include run-time validation. Predefined standard forms of validation



conjectures have been defined, directly supporting the validation of a deadline or separation between two events, called the *trigger* and the *response*. The trigger event could be the press of a button, and the corresponding response may be the update of a display. From the standard forms, more specific validation conjectures can be constructed.

In Overture it is possible to specify validation conjectures as comments in the **system** class. Listing 15.1 provides an example of a validation conjecture requiring the separation of 500 ms between two subsequent screen updates.

```
system Distribution
... // Class content omitted

/* timing invariants
separate(#fin(MMI`UpdateScreen), #fin(MMI`UpdateScreen), 500 ms);
*/

end Distribution
```

Listing 15.1: Validation conjecture example


The concrete syntax for the timing invariants in VDM-RT is defined as:

```
property(trigger, ending, interval);
```

The different kinds of properties that can be used are called:

deadlineMet: A deadline by definition is a time by which something must be finished. In real-time embedded systems there is typically deadlines that must be respected from when an event happens to its response. In our terminology, it means that the ending event must happen within a certain timeframe from the trigger event.

seperate: The separation properties are used to describe a minimum separation between events if the second event occur at all.

In Overture validation conjectures are evaluated over the execution trace at run-time and in case violations occur they will be written to a *violations file* with `.txt` extension. The violations file is located in the log file directory and named by the time of execution. When this file is loaded using the  button, Overture will list the violations as shown in Figure 15.4. In particular, this figure shows two violations of the validation conjecture in listing 15.1.

In addition, the RealTime Log viewer will show the violations graphically by marking the trigger and corresponding response in the model execution overview. Figure 15.5 shows the two violations from Figure 15.4. Note that the trigger for the second violation occurs at the same time as the response for the first one, Therefore, they are both marked by the same red circle (the middle one).



status	name	expression	src time	src thread	dest time	dest thread
FAIL	C1	separate(#fin(MMIUpdateScreen),#fin(MMIUpdateScreen),500000000)	118254267	14	209165360	15
FAIL	C1	separate(#fin(MMIUpdateScreen),#fin(MMIUpdateScreen),500000000)	209165360	15	300076453	16

Figure 15.4: Violations of validation conjectures will be listed in a table

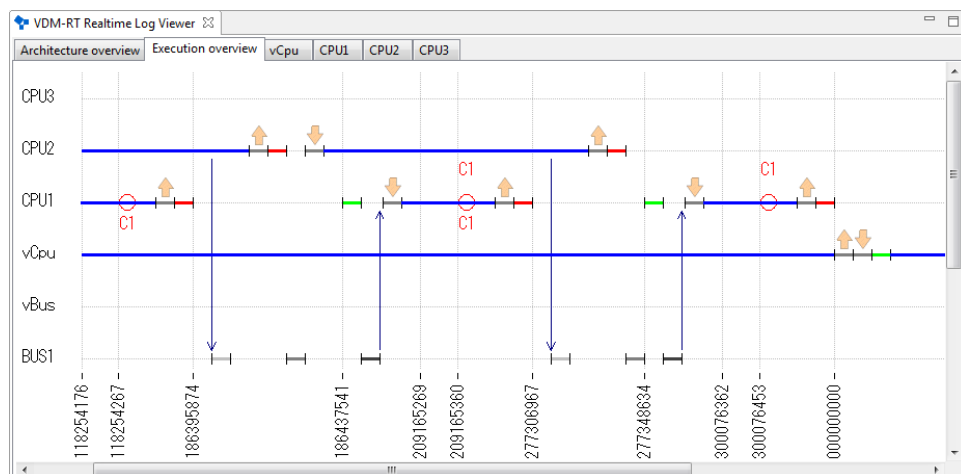


Figure 15.5: Violations of validation conjectures shown in the model execution overview

Chapter 16

Expanding VDM Models Scope and Functionality by Linking Java and VDM

In some cases the impact and value of VDM models can be immensely improved by expanding them with functionality which is not delivered directly by VDM. Examples of such functionality could be; (a) to associate the model with existing legacy systems, for which no model or specification exists, (b) to add a graphical representation of the model, or (c) to enable network communication, for instance in a client-server setup. In order to achieve this functionality Overture enables the possibility of linking a VDM model directly with the underlying Java-based Overture interpreter. In addition it is possible to link a Java implementation with a VDM model.

In order to co-execute VDM and Java, the Java build path must be configured as explained in Section 16.1. Overture supplies two different techniques for linking between VDM and Java; (1) the functionality which allow VDM to interact with an “External Java Library” is described in Section 16.2 and (2) the method used to allow a Java implementation to “Remote Control” a VDM model is explained in Section 16.3. An example of how these methods can be used to create a GUI for a model is supplied in Section 16.4. These methods are also used to enable control of models with a generic GUI as explained in section 16.5.

16.1 Configuring the Java Build Path

In order to use Overture’s implementation of the different VDM types in a Java program, the Overture library must be added to the Java program’s build path. The library can be found in the Overture installation directories under:

`<Overturedir>\Plugins\:`

- `org.overture.ide.core:x.x\jars,`

Ast.jar Contains the Abstract Syntax Tree with all the static language definitions

Parser.jar Contains the VDM parser



- `org.overture.ide.debug:_x.x\jars\`

Interpreter.jar Contains the interpreter and the values which is what external Java implementations mainly will use)

- `org.overture.ide.builder.vdmj:_x.x\jars\`

TypeChecker.jar Contains the type checker, this is used when e.g. comparing types

These can then be added to the Java build path, as shown in Figure 16.1. The alternative is to use the command line `Overture.x.x.x.Jar` file, as described in chapter 17, and add that to the Java build path, since this contains all of the above Jars in a single file.



Figure 16.1: Adding the Overture library to the build path in Eclipse

16.2 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` file¹. 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].

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



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.

16.2.1 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 16.1 and the Java class implementing it can be seen in listing 16.2. The values that are to be exchanged between the Overture IDE and the jar file needs to be the internal *Value* objects used in Overture. Documentation about these classes can be found in Appendix I.

```
class remote_lib_sensor

operations

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

end remote_lib_sensor
```

Listing 16.1: Remote sensor VDM class

```
package remote.lib;

import org.overture.interpreter.runtime.ValueException;
import org.overture.interpreter.values.IntegerValue;
import org.overture.interpreter.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 16.2: Remote sensor Java class



16.3 Enabling Remote Control of the Overture Interpreter

In some situations, it may be valuable 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.

16.3.1 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 16.3, 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 Overture. Documentation about these can be found in the Overture Design Specification [Battle10].

```
import org.overture.interpreter.debug.RemoteControl;
import org.overture.interpreter.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 16.3: Remote Controller Java class



16.4 Using a GUI in a VDM model: Linking Example

This example describes how the linking functionality can be used to create a graphical representation of a VDM model.

16.4.1 The Modelled System

A GUI has been developed in Java Swing which will be used to both control and present the system which is described in the VDM model. In this example both the “External Java Library” technique as well as the “Remote Controller” technique is utilized. This is done to enable the VDM model to display data directly to the Java GUI, and to allow the VDM model to be controlled from the Java GUI.

The example is based on a VDM++ model of the smokers concurrency problem [Patil71]. Consider a scenario where three chain smokers and an agent, who does not smoke, are sitting at a table, infinitely going through the following lifecycle:

1. Each chain smoker continuously seeks to roll a cigarette and smoke it,
2. a smoker needs three ingredients: tobacco, paper and matches,
3. each smoker has an infinite supply of only one of the ingredients. One of the smokers has tobacco, the second has paper, and the third has matches,
4. the agent has an infinite supply of all three materials and randomly places two different ingredients on the table at a time,
5. the smoker who has the remaining ingredient then empties the table, rolls a cigarette and smokes it. The smokers never accumulate the ingredients and never grab an ingredient from the table which they are already in possession of,
6. when the table becomes empty, the agent puts another two random ingredients on the table, and the cycle repeats.

In this example a GUI has been created for the model in which the user of the GUI is considered to be the agent providing the smoker with ingredients, as illustrated in Figure 16.2. By clicking one of three buttons one of the respective ingredients will be placed on the table. Once the necessary ingredients are on the table, one smoker will grab them and start smoking.

The architecture of the example is illustrated on Figure 16.3, with a focus on the linking functionality. The diagram shows central classes and relations, and it places the different classes into a GUI, a Java and a VDM/Overture block. In the given context the *World* class represents the entire VDM model. In the diagram color highlights are used to distinguish the two linking techniques.



Figure 16.2: GUI of Smokers Example

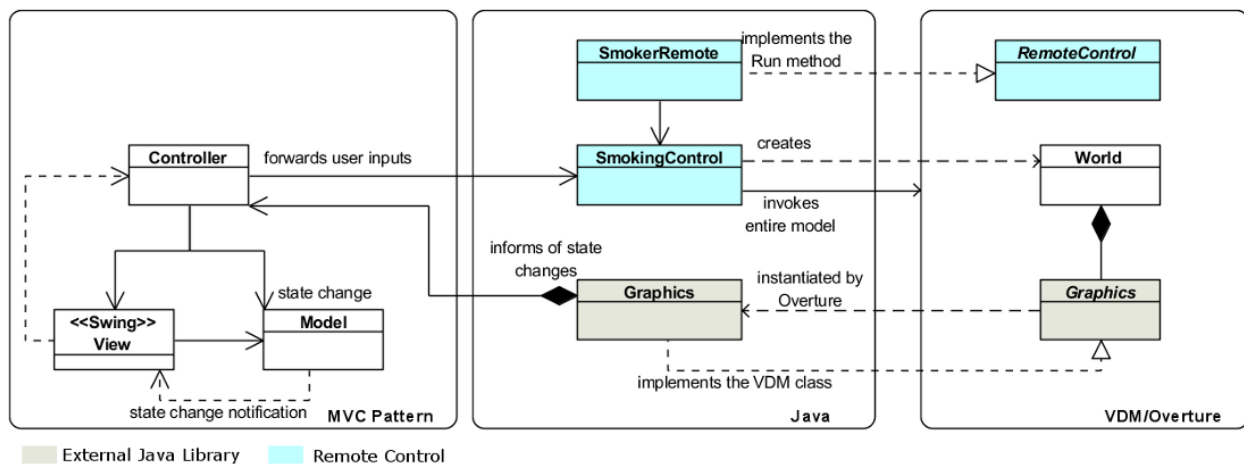


Figure 16.3: Class diagram of the Smokers example architecture

16.4.2 External Java Library

To allow the GUI to be a true graphical representation of the model, the model itself will be able to update the GUI whenever it needs to by using the “External Java Library” technique. If only the remote controller was used, the model would be unable to update the GUI and it would rely on the



Java implementation to request data on any state changes in the model.

A VDM model interface has been created which allows the model to interact with the Java GUI, the interface is shown in Listing 16.4. The hierarchical naming pattern of Java packages, which normally are separated by periods (.), are denoted with underscores in the VDM model. Meaning that with this interface the External Java library must contain a *Graphics* class which is organized in the *gui* package.

```
class gui_Graphics
operations

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

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

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

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

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

  public nowSmoking : nat ==> ()
    nowSmoking(smokerNumber) == is not yet specified;
end gui_Graphics
```

Listing 16.4: VDM interface for external Java library

The Java class implementing the VDM interface is shown in Listing 16.5. This example will not go into detail with regards to actual GUI implementation, however it should be mentioned that the example utilizes the Model-View-Controller (MVC) design pattern and that the Java implementation of the VDM interface interacts with the graphical representation through the *model* object (this is the *model* in the MVC pattern and not a representation of the VDM model).

It should be noted that the names of the package and class can be directly related to VDM interface, i.e. *gui* and *Graphics*. Furthermore the imports should be noted; firstly the class must be marked as *Serializable*, secondly multiple packages from the Overture interpreter are imported as well. These are needed for the conversion between VDM values and Java values, as it can be seen in the *nowSmoking* method in Listing 16.5.

Please be aware that it is extremely important that there are no unused imports in the Java implementation, as this will result in an error when the Java library is loaded by Overture.



```
package gui;

import java.io.Serializable;
import org.overture.interpreter.runtime.ValueException;
import org.overture.interpreter.values.Value;
import org.overture.interpreter.values.VoidValue;

public class Graphics implements Serializable {

    Controller ctrl;
    Model model;

    public Value init() {
        ctrl = new Controller(); //init the Controller of the MVC pattern
        model = ctrl.getModel();
        return new VoidValue();
    }

    public Value tobaccoAdded() {
        model.tobaccoAdded();
        return new VoidValue();
    }

    public Value nowSmoking(Value smokeid) throws ValueException {
        model.nowSmoking(smokeid.intValue(null)); //set smoker
        ctrl.DisableButtons(); //prevent new GUI input
        model.finishedSmoking(); //wait for smoker to finish
        ctrl.EnableButtons(); //enable GUI input
    }
    ...
}
```

Listing 16.5: Java implementation of the VDM interface for the external Java library



16.4.3 Remote Control

To enable the GUI to interact with the model the “Remote Control” technique is utilized, by implementing the *RemoteControl* interface, as shown in Listing 16.6. The *RemoteControl* interface also requires the Overture Java library to be included in the Java build path, as explained above. From listing 16.6 it can be seen that the *RemoteControl* interface supplies protected access to the VDM interpreter, that is passed to the *SmokingControl* object which functions as the bridge between the GUI and the running VDM model. The *SmokingControl* class is specific for this example, and its implementation could essentially be implemented directly in the *RemoteControl* realization.

It is important the *finish* method is called on the interpreter when the GUI is disposed, this will allow Overture to do a controlled shut-down of the remote interpretation, keeping the debugger alive for post execution communication such as coverage writing.

```
public class SmokerRemote implements RemoteControl {

    RemoteInterpreter interpreter;
    @Override
    public void run(RemoteInterpreter intrprtr) throws Exception {

        interpreter = intrprtr;
        SmokingControl ctrl = new SmokingControl(interpreter);
        ctrl.init();
    }
}
```

Listing 16.6: Java implementation of the RemoteControl interface

The implementation of the *SmokingControl* is shown in Listing 16.7. In the *init* method it can be seen how VDM statements are executed as strings commands, the *World* class is created and the *Run* operation is invoked. This is the basic way of interacting with a model through the remote control functionality. The *finish* method informs the interpreter that the remote GUI is being disposed and that execution should be stopped. The *AddPaper* method shows how the variables defined in the *init* method can be used for invoking an operation. The *AddPaper* method is called directly from the GUI Button click action, as shown in Listing 16.8.

```
public class SmokingControl {

    RemoteInterpreter interpreter;
    public SmokingControl(RemoteInterpreter intrprtr) {
        interpreter = intrprtr;
        Controller.smoke = this;
    }

    public void init() {
        interpreter.create("w", "new World()");
        interpreter.valueExecute("w.Run()");
    }

    public void AddPaper() {
```



```
        interpreter.valueExecute("w.agent.AddPaper()");
    }

    public void finish(){
        interpreter.finish();
    }
    ...
}
```

Listing 16.7: Java implementation of the bridge between the GUI and the interpreter executing the VDM model

```
addPaperButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        smoke.AddPaper();
    }
});
```

Listing 16.8: Java implementation of the Button click action invoking the remote interpreter

The last thing to note in Listing 16.7 is the constructor which adds itself to a public static field in the *Controller* class. This step is necessary to bridge the objects created by the “Remote Control” technique with the objects created in connection with the “External Java Library” technique. To understand why this construct is needed, some insight into the initialization steps is necessary. Firstly the entire VDM model, and thereby the *Graphics* object, is created via the *RemoteControl* interface by *SmokingControl*. Now recall from Listing 16.5 that the *Controller* class, of the MVC pattern, is created when the *init* method of the *Graphics* class is invoked from the VDM model, meaning that the actual graphical Java components, are also created when *init* is called. Now in order to connect a click on the GUI buttons with the commands that can be executed in the model, the *Controller* needs to have access to the interpreter, i.e. the *SmokingControl* object. Meanwhile the *Graphics* object is created deep inside the VDM model and the *SmokingControl* object cannot be passed to it through the interpreters execute method. Instead the bridge between the *Controller* and the *SmokingControl* object is kept in Java, and the insider knowledge that the *Controller* object will eventually be created from inside the model, is used to justify the static reference. This is a special case when combining the “Remote Control” technique with the “External Java Library” method.

Example of how to shut-down a JFrame when using the remote controller interface

It is important that the execution of a remote GUI is stopped in a controlled manner e.g. not just by calling `System.exit(0)`. One way to allow the finish method to be called from a `JFrame` is shown in listing 16.9; The in the constructor of the `JFrame` a the default close operation is changed to be `DISPOSE_ON_CLOSE`, this will change the closing of the window so that the `dispose` method is called where call to the interpreter finish can be placed.

```
public MyJFrame() {
    // Allow Overture to do a controlled shutdown
```




```
setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);  
...  
}  
...  
@Override  
public void dispose() {  
    interpreter.finish();  
}
```

Listing 16.9: Java implementation of a finish method for a JFrame.

16.4.4 Deployment of the Java Program to Overture

Once the Java program has been implemented, it must be exported to a Jar file and placed in the lib directory in the Overture/VDM projects directory, in order for Overture to find it. In Eclipse a Jar file can be created through the Export function, as illustrated on Figures 16.4 and 16.5.

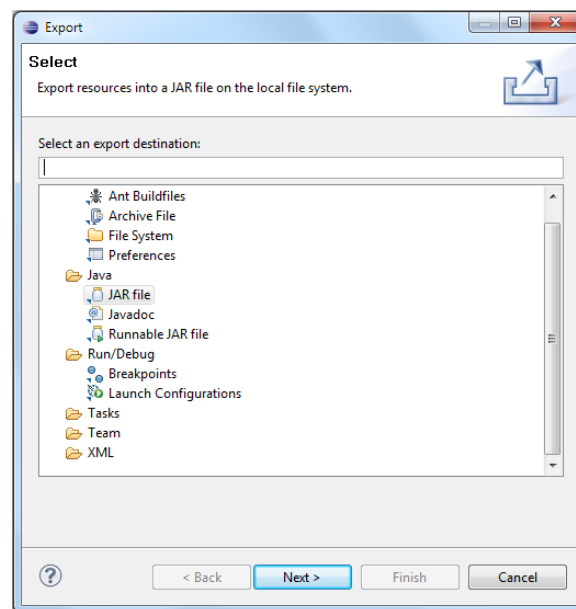


Figure 16.4: Exporting in Eclipse

Note that it can be very beneficial to check “Save the description of this JAR”, illustrated on Figure 16.6, as this saves the export configuration and makes it a lot faster to redeploy the JAR file to VDM project during development.

Before running the model in Overture the debug configuration needs to be changed to use the Remote Control. “The Launch Mode” must be change to “Remote Control”, and the fully qualified name of the class implementing the remote control interface must be supplied. The debug configuration for the current example is supplied in Figure 16.7.

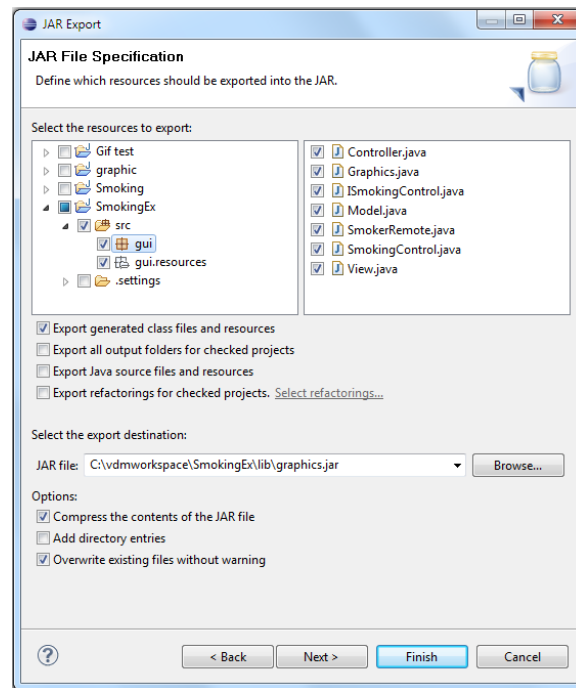


Figure 16.5: Exporting to a Jar in Eclipse

16.5 Generated GUI of VDM Models

Overture has a feature for controlling a VDM model with a generic GUI, automatically generated from the model [Nunes&2011].² The generated GUI follows the same principles as discussed in section 16.4. But rather than the user having to implement the GUI himself, everything is automatically generated. In order to use the Generated GUI feature, simply launch the model as a VDM GUI configuration (see Figure 16.8).

Once the model has launched, the generic GUI will offer a list of all classes allowing to create instances of each (see Figure 16.9). These instances can then be selected to inspect their state and invoke operations (see Figure 16.10). Any created instance or operation called will be sent to the model, allowing to interact with it. Any responses from the model are also shown in the GUI.

²This feature is currently only available for VDM++ models.

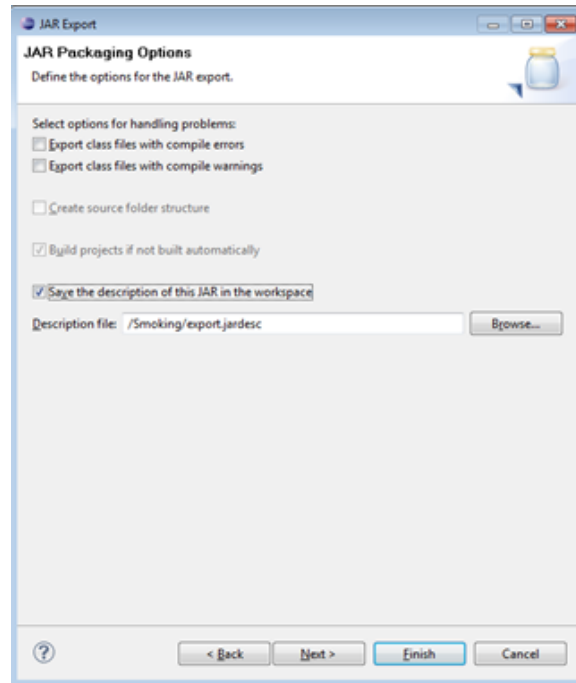


Figure 16.6: Saving the description of the export for future use

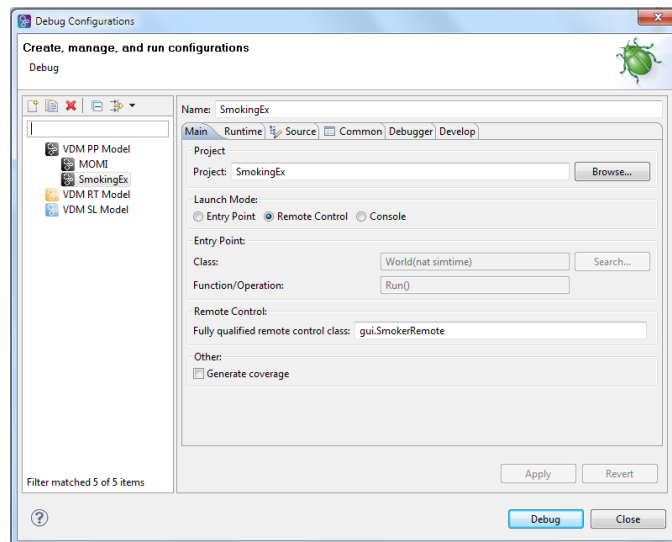


Figure 16.7: Changing the Overture debug configuration into using the remote controller

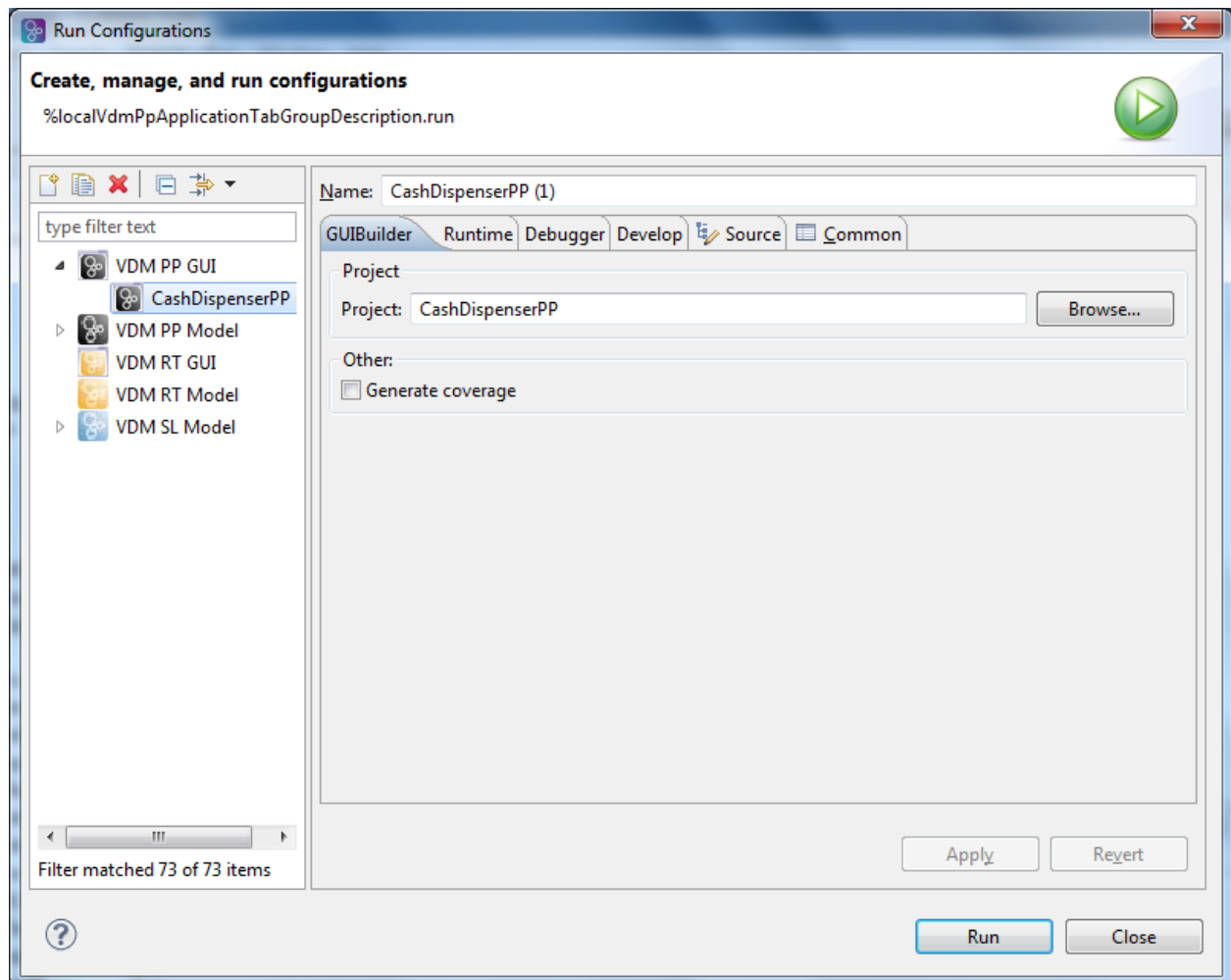


Figure 16.8: VDM GUI Launch configuration

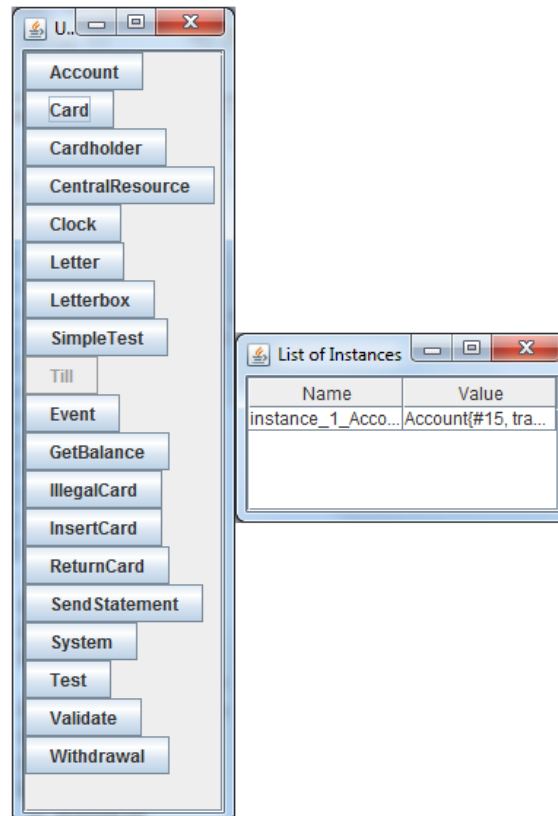


Figure 16.9: Automated GUI class list for Cash Dispenser example.

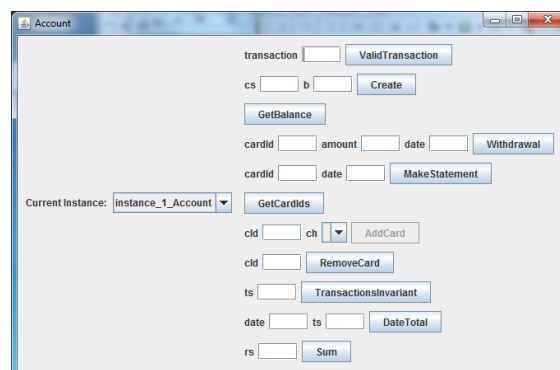


Figure 16.10: Instance viewer showing the Account class.



Chapter 17

A Command-Line Interface to Overture

At the centre of the Overture tool there is a Java implementation of VDM forming a core. This provides a command-line interface that may be valuable as it can be used independently of the Eclipse interface of Overture. The command-line Jar is both bundled with the Overture tool, located in the `commandline` folder, and available as a standalone download.

17.1 Starting Overture at the Command-Line

The `Overture-x.x.x.jar` file contains a MANIFEST that identifies the main class to start the tool, so the minimum command line invocation is as follows:

```
$ !\textbf{java -jar Overture-x.x.x.jar}!  
Overture: You must specify either -vdmsl, -vdmpp or -vdmrt  
Usage: Overture <-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 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 15).
- remote:** This enables remote control of the Overture executable.

Alternatively, a script file will also be made for the different platforms (script and bat files) wrapping this functionality so one does not need to explicitly point to the Overture jar. This script is planned for the next version version of the tool.

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 Overture will load all files in that directory with a suffix that matches the dialect (e.g. `*.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.

17.2 Parsing, Type Checking, and Proof Obligations Command-Line

All specification files loaded are parsed and type checked automatically by the command-line tool. 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 Overture 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.

17.3 The Command-Line 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:

```
$ !\textbf{java -jar Overture-x.x.x.jar -vdmsl -i shmem.vdmsl}!
Parsed 1 module in 0.14 secs. No syntax errors
Warning 5000: Definition 'i' not used in 'M' (shmem.vdmsl)
  at line 129:7
Type checked in 0.078 secs. No type errors
Initialized 1 module in 0.046 secs.
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_VARVALUES” 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 L^AT_EX coverage files. These are L^AT_EX versions of the source files with parts of the specification highlighted where they have not been executed. The L^AT_EX 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 L^AT_EX 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.

```
> !\textbf{modules}!  
M (default)
```

```
> !\textbf{state}!  
Q4 = [mk_M(<FREE>, 0, 9999)]  
rseed = 87654321  
Memory = mk_Memory(87654321, [mk_M(<FREE>, 0, 9999)],  
                           [mk_M(<FREE>, 0, 9999)])  
Q3 = [mk_M(<FREE>, 0, 9999)]
```

```
> !\textbf{print rand(100)}!  
= 71  
Executed in 0.063 secs.
```

```
> !\textbf{print rand(100)}!  
= 44  
Executed in 0.0 secs.
```

```
> !\textbf{state}!  
Q4 = [mk_M(<FREE>, 0, 9999)]  
rseed = 566044643  
Memory = mk_Memory(566044643, [mk_M(<FREE>, 0, 9999)],  
                           [mk_M(<FREE>, 0, 9999)])  
Q3 = [mk_M(<FREE>, 0, 9999)]
```

```
> !\textbf{init}!  
Global context initialized
```

```
> !\textbf{state}!  
Q4 = [mk_M(<FREE>, 0, 9999)]  
rseed = 87654321  
Memory = mk_Memory(87654321, [mk_M(<FREE>, 0, 9999)],  
                           [mk_M(<FREE>, 0, 9999)])
```



```
Q3 = [mk_M(<FREE>, 0, 9999)]
```

```
> !\textbf{env}!
fragments = (M`Quadrant -> nat)
combine = (M`Quadrant -> M`Quadrant)
tryBest = (nat ==> nat)
seed = (nat1 ==> ())
reset = (() ==> ())
bestfit = (nat1 * M`Quadrant -> nat1)
add = (nat1 * nat1 * M`Quadrant -> M`Quadrant)
firstFit = (nat1 ==> bool)
rand = (nat1 ==> nat1)
tryFirst = (nat ==> nat)
main = (nat1 * nat1 ==> seq of (<SAME> | <BEST> | <FIRST>))
MAXMEM = 10000
delete = (M`M * M`Quadrant -> M`Quadrant)
inv_M = (M`M +> bool)
CHUNK = 100
bestFit = (nat1 ==> bool)
least = (nat1 * nat1 -> nat1)
fits = (nat1 * M`Quadrant -> nat1)
init_Memory = (M`Memory +> bool)
pre_add = (nat1 * nat1 * M`Quadrant +> bool)
```

This example shows a VDM-SL specification called `shmem.vdmsl` being loaded. The help command lists the interpreter commands available. Note that several of them regard the setting of breakpoints, which is covered in the next section.

The **state** command lists the content of the default module's state. This can be changed by operations, as can be seen by the two calls to `rand` which change the `seed` value in the state (a pseudo-random number generator). The `init` command will re-initialize the state to its original value, illustrated by the fact that two subsequent calls to `rand` return the same results as the first two did.

The `print` command can be used to evaluate any expression. The `env` command lists all the values in the global environment of the default module. This shows the functions, operations and constant values defined in the module. Note that it includes invariant, initialization and pre/postcondition functions. The `pog` command (proof obligation generator) generates a list of proof obligations for the specification.

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 SourceForge 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.
- [Burdy&05] Lilian Burdy and Yoonsik Cheon and DavidR. Cok and Michael D. Ernst and Joseph R. Kiniry and Gary T. Leavens and K. Rustan M. Leino and Erik Poll. An overview of JML Tools and Applications. *Intl. Journal of Software Tools for Technology Transfer*, 7:212–232, 2005.
- [CGMan] The VDM Tool Group. *The VDM-SL to C++ Code Generator*. Technical Report, CSK Systems, January 2008.



- [CGManPP] The VDM Tool Group. *The VDM++ to C++ Code Generator*. Technical Report, CSK Systems, January 2008.
- [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 *Safecom’99*, Springer Verlag, Toulouse, France, September 1999. LNCS 1698, ISBN 3-540-66488-2.
- [DelegateTutorial] Delegate Tutorial Github project. <https://github.com/ldcouto/delegate-tutorial>, 2016. pages. .
- [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&07] John Fitzgerald and Peter Gorm Larsen and Simon Tjell and Marcel Verhoef. *Validation Support for Distributed Real-Time Embedded Systems in VDM++*. Technical Report CS-TR:1017, School of Computing Science, Newcastle University, April 2007. 18 pages.
- [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&14] John Fitzgerald and Peter Gorm Larsen and Marcel Verhoef, editors. *Collaborative Design for Embedded Systems – Co-modelling and Co-simulation*. Springer, 2014.
- [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.
- [Java2VDMMan] The VDM Tool Group. *The Java to VDM++ User Manual*. Technical Report, CSK Systems, January 2008.
- [Johnson96] C.W. Johnson. Literate Specifications. *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):343–355, October 2009. 13 pages.
- [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 Embedded Systems using VDM. *Intl. Journal of Software and Informatics*, 3(2-3), October 2009.
- [Larsen&10] Peter Gorm Larsen and Kenneth Lausdahl and Nick Battle. *The VDM-10 Language Manual*. Technical Report TR-2010-06, The Overture Open Source Initiative, April 2010.
- [Larsen&13] Peter Gorm Larsen and Kenneth Lausdahl and Nick Battle and John Fitzgerald and Sune Wolff and Shin Sahara. *VDM-10 Language Manual*. Technical Report TR-001, The Overture Initiative, www.overturetool.org, April 2013. 208 pages.
- [Larsen&96] Peter Gorm Larsen and Bo Stig Hansen. Semantics for Underdetermined Expressions. *Formal Aspects of Computing*, 8(1):47–66, January 1996.
- [Lucas78] P. Lucas. On the Formalization of Programming Languages: Early History and Main Approaches. In *The Systematic Development of Compiling Algorithms*, 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.
- [Nunes&2011] Nunes, Carlos and Paiva, Ana. Automatic Generation of Graphical User Interfaces From VDM++ Specifications. In *ICSEA 2011, The Sixth International Conference on Software Engineering Advances*, pages 399–404, 2011.



- [Patil71] S. S. Patil. *Limitations and Capabilities of Dijkstra's Semaphore Primitives for Coordination among Processes*. Cambridge, Mass.: MIT, Project MAC, Computation Structures Group Memo 57, February 1971.
- [Ribeiro&11] Augusto Ribeiro and Kenneth Lausdahl and Peter Gorm Larsen. Run-Time Validation of Timing Constraints for VDM-RT Models. In Sune Wolff and John Fitzgerald, editors, *Proceedings of the 9th Overture Workshop*, pages 4–16, June 2011.
- [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, Springer-Verlag, 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



Key	Description
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



Key	Description
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(period, 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://github.com/overturetool/overture/issues/new>).

- 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"

0067: "Exported type <name> not structured"

0068: "Periodic threads overlapping"

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' or 'in seq' after pattern in binding"
- 2001:** "Expecting 'in set' or 'in seq' in bind"
- 2002:** "Expecting ':' in type bind"
- 2003:** "Expecting 'in set' or 'in seq' after pattern in binding"
- 2004:** "Expecting 'in set', 'in seq' 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:** "-"
- 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"



APPENDIX D. SYNTAX ERRORS

- 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"



APPENDIX D. SYNTAX ERRORS

- 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|seq <exp>"
- 2111:** "Expecting <pattern> in set|seq <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"



APPENDIX D. SYNTAX ERRORS

- 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' "



APPENDIX D. SYNTAX ERRORS

- 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"



APPENDIX D. SYNTAX ERRORS

- 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"
- 2296: "Block cannot be empty"
- 2297: "Expecting '|->' in map pattern"
- 2298: "Map patterns not available in VDM classic"
- 2299: "Expecting {|->} empty map pattern"



APPENDIX D. SYNTAX ERRORS

- 2300:** "mk_<type> must have a single argument"
- 2301:** "Expecting narrow_(expression, type) "
- 2302:** "Expecting ')' after narrow_ expression"
- 2303:** "Narrow not available in VDM classic"
- 2304:** "'stop' not available in VDM classic"
- 2305:** "'stoplist' not available in VDM classic"
- 2306:** "Expecting 'stop' "
- 2307:** "Expecting 'stop(' "
- 2308:** "Expecting ')' after stop object"
- 2309:** "Expecting 'stoplist' "
- 2310:** "Expecting 'stoplist(' "
- 2311:** "Expecting ')' after stoplist objects"
- 2312:** "Expecting '(' after sporadic"
- 2313:** "Expecting ')' after sporadic arguments"
- 2314:** "Expecting '(' after sporadic(...)"
- 2315:** "Expecting (name) after sporadic(...)"
- 2316:** "Periodic threads only available in VDM-RT"
- 2317:** "Sporadic threads only available in VDM-RT"
- 2318:** "Unexpected token after flat definitions"
- 2319:** "Expecting class name after obj_ in object pattern"
- 2320:** "Expecting '(' after obj_ pattern"
- 2321:** "Expecting '|->' in object pattern"
- 2322:** "Expecting ')' after obj_ pattern"
- 2323:** "Object patterns not available in VDM classic"
- 2324:** "Pure only permitted for operations"



2325: "Pure operations are not available in classic"

2326: "Expecting 'of' after set1"

2327: "Type set1 is not available in classic"

2328: "Sequence binds are not available in classic"

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/seq bind is not a set/seq type?"
- 3016:** "Expression is not compatible with set/seq 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"



APPENDIX E. TYPE ERRORS AND WARNINGS

- 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"



APPENDIX E. TYPE ERRORS AND WARNINGS

- 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/seq bind is not a set/seq"
- 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"



APPENDIX E. TYPE ERRORS AND WARNINGS

- 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 '^' is not a sequence"
- 3158:** "Right hand of '^' is not a sequence"



- 3159:** "Predicate is not boolean"
- 3160:** "Left hand of ' \setminus ' is not a set"
- 3161:** "Right hand of ' \setminus ' is not a set"
- 3162:** "Left and right of ' \setminus ' 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"



APPENDIX E. TYPE ERRORS AND WARNINGS

- 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/seq bind is not a set/seq"
- 3198:** "Type bind not compatible with expression"
- 3199:** "Set/seq 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"



APPENDIX E. TYPE ERRORS AND WARNINGS

- 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/seq"
- 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 nat"
- 3282:** "Arguments to cycles must be nat"
- 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"



APPENDIX E. TYPE ERRORS AND WARNINGS

- 3286:** "Constructor cannot be 'async'"
- 3287:** "Periodic/sporadic thread must have <n> argument(s)"
- 3288:** "--"
- 3289:** "--"
- 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 too slow: <speed> Hz"
- 3306:** "CPU frequency too fast: <speed> Hz"
- 3307:** "Errs clause is not bool -> bool"
- 3308:** "Cannot mix modules and flat specifications"
- 3309:** "Measure must not be polymorphic"
- 3310:** "Measure must also be polymorphic"



- 3311:** "Pattern cannot match"
- 3312:** "Void operation returns non-void value"
- 3313:** "Operation returns void value"
- 3314:** "Map pattern is not matched against map type"
- 3315:** "Matching expression is not a map type"
- 3316:** "Expecting number in periodic/sporadic argument"
- 3317:** "Expression can never match narrow type"
- 3318:** "Measure's type parameters must match function's"
- 3319:** "'in set' expression is always false"
- 3320:** "'not in set' expression is always true"
- 3321:** "Type component visibility less than type's definition"
- 3322:** "Duplicate patterns bind to different types"
- 3323:** "Overloaded operation cannot mix static and non-static"
- 3324:** "Operation <name> is not static"
- 3325:** "Mismatched compose definitions for <type>"
- 3326:** "Constructor can only return 'self'"
- 3327:** "Value is not of the right type"
- 3328:** "Statement may return void value"
- 3329:** "Abstract function/operation must be public or protected"
- 3330:** "Cannot instantiate abstract class <name>"
- 3331:** "obj_ expression is not an object type"
- 3332:** "Object pattern cannot be used from a function"
- 3333:** "Matching expression is not a compatible object type"
- 3334:** "<name> is not a matchable field of class <class>"
- 3335:** "Subset will only be true if the LHS set is empty"



APPENDIX E. TYPE ERRORS AND WARNINGS

3336: "Illegal use of RESULT reserved identifier"

3337: "Cannot call a constructor from here"

3350: "Polymorphic function has not been instantiated"

3351: "Type parameter '<name>' cannot be used here"

3352: "Exported <name> function has no type paramaters"

3353: "Exported <name> function type parameters incorrect"

3354: "Function argument must be instantiated"

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/sequence 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"

5016: "Some statements will not be reached"

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:

VDMJ Error and Warning Messages

- 4000:** "Cannot instantiate abstract class <class>"
- 4002:** "Expression value is not in set/seq 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/seq bind does not contain value <value>"

4040: "Let be st found no applicable bindings"

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



APPENDIX F. RUN-TIME ERRORS

- 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/seq bind"
- 4050:** "Value <value> is not in set/seq 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>"



APPENDIX F. RUN-TIME ERRORS

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"



APPENDIX F. RUN-TIME ERRORS

- 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"
- 4151:** "Cannot take dinter of empty set"
- 4152:** "Wrong number of elements for map pattern"
- 4153:** "Values do not match map pattern"
- 4154:** "Cannot match map pattern"
- 4155:** "Map union pattern does not match expression"
- 4156:** "Cannot match map pattern"
- 4157:** "Expecting +ive integer in periodic/sporadic argument <n>"
- 4158:** "Period argument must be non-zero"
- 4159:** "Delay argument must be less than the period"
- 4160:** "Object <#n> is not running a thread to stop"
- 4161:** "Cannot stop object <#n> on CPU <name> from CPU <name>"
- 4162:** "The RHS range is not a subset of the LHS domain"
- 4163:** "Cannot inherit private constructor"
- 4164:** "Compose function cannot be restricted to <type>"
- 4165:** "Cannot convert <type> to <type>"
- 4168:** "Arguments do not match parameters: <name(params)>"
- 4170:** "Cannot convert empty set to set1"



Appendix G

Categories of Proof Obligations

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

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.

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.

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.

function compose: When using a function composition (`f comp g`), this ensures that the precondition of `g` implies the precondition of `f` applied to the result of `g`.

function iteration: When using a function iteration, for the function we are iterating with, this ensures that the precondition on an argument implies the precondition on the result.

function parameter patterns: When using a pattern as a function parameter, this ensures that all values in the parameter type for the function can match the pattern.

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.



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.

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

map compose: When composing 2 maps, ensures that the range of map2 is a subset of the domain of map1.

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 iteration: When performing a map iteration, ensures the iteration count expression is either 0 or 1 or if it's greather than 1 then the map's range is a subset of its domain.

map sequence compatible: When defining a map with enumeration, ensures that any two equal elements in the domain map to the same element in the range.

map set compatible: When merging a set of maps, any two equal elements in the domains of each map map to the same element in the range.

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

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

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

operation parameter patterns: When using a pattern as an operation parameter, ensures that all values in the operation parameter type can match the pattern.

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 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.

post condition: Whenever a function has a post condition this checks that the precondition of the function implies the post condition.

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

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



sequence modification: Whenever a sequence modification is used, this ensures the domain of the modification map is a subset of the indices of the sequence.

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.

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

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

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

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

value binding: When binding a value to a pattern, ensures that the resulting value matches the pattern.

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



Appendix H

Mapping Rules between VDM++/VDM-RT and UML Models

Transformation Rule 1

VDM classes are mapped as the UML meta-class `Class`

Transformation Rule 2

The visibility of VDM instance variables, values, functions and operations are mapped as a *subset* of the UML enumeration `VisibilityKind` comprising `public`, `private` and `protected`.

Transformation Rule 3

VDM `static` is mapped as the `isStatic` property of the UML meta-class `Class`, `Property` or `Operation` respectively.

Transformation Rule 4

Data type definitions are mapped as the UML meta-class `Class` and are referenced, and thus nested, through the meta-attribute `nestedClassifier` of the owning class. Notice that this rule is not specified or implemented.

Transformation Rule 5

Instance variable and value definitions are mapped as the UML meta-class `Association`, if:

5 a: The type is an *object reference type*, or

5 b: The type is *not* a basic *data type* [Fitzgerald&05, p64,71].

**Transformation Rule 6**

Instance variable and value definitions are mapped as the UML meta-class `Property`, if the type is a *basic data type* [Fitzgerald&05, p71]. Instance variables and values are distinguished by the meta-attribute `isReadOnly`. Notice: rule 10 is an exception to this rule.

VDM concept	<code>Property::isReadOnly</code>
Instance variables	false
Values	true

Table H.1: The meta-attribute `isReadOnly` distinguishes instance variables and values

Transformation Rule 7

The initial value of instance variables and values definitions are mapped as the property default of the UML meta-class `Property`.

Transformation Rule 8

The VDM optional type is mapped to the properties `lower = 0` and `upper = 1` of the UML meta-class.

Transformation Rule 9

The VDM constructs `set`, `seq` and `seq1` is mapped as the UML meta-class `Association` which may be decorated with a textual constraint defined by the meta-attribute `isOrdered`¹ in addition to a multiplicity at both ends. Table H.2 shows how the above-mentioned VDM constructs are mapped.

VDM construct	Ordered	Target class Multiplicity
<code>set</code>	false	0..*
<code>seq</code>	true	0..*
<code>seq1</code>	true	1..*

Table H.2: Transformation rules for VDM constructs modeling collections

**Transformation Rule 10**

The VDM constructs `map` and `inmap` are mapped as the UML meta-class `Association` with a qualifier. The domain is specified by the qualifier, which is located at the source class. The range is specified by the target class. Notice, that if the range is specified by a *basic type* it is mapped as a separate class. This is an exception to rule 6.

VDM construct	Qualifier end <code>isUnique</code>	Target class end <code>isUnique</code>
<code>map</code>	<code>false</code>	<code>true</code>
<code>inmap</code>	<code>true</code>	<code>true</code>

Table H.3: Transformation rules for VDM constructs modeling relationships between two sets.

Transformation Rule 11

A VDM class with a thread compartment is mapped as the UML meta-class `Class` with the meta-attribute `isActive` set to `true`.

Transformation Rule 12

A VDM class with the keyword `is subclass of` followed by class-names is mapped as the UML meta-class `Generalization`, with the attributes `general` and `specific` referencing the superclass and subclass, respectively. More than one subclass results in more than one instance of `Generalization`.

Transformation Rule 13

A VDM class with the keyword `is subclass responsibility as` a function or operation body is mapped as the UML meta-class `Class` with the meta-attribute `isAbstract` set to `true`.

Transformation Rule 14

A VDM generic class maps to the UML meta-class `Class` with the attribute `templateSignature` referencing a set of `TemplateParameter` having the name property set to the name of the parameter.

**Transformation Rule 15**

A VDM operation and function are mapped to the UML meta-class `Operation` where the property `isQuery` determine whether the `Operation` represents a VDM function or operation:

true

- for a function.

false

- for a operation.

The return type of a function and operation is mapped collectively as the property `type` and the `multiplicity`² of the `Operation` meta-class. The parameters of the operation or function is mapped to the UML meta-class `Parameter` represented as the property `ownedParameters` of the `Operation` meta-class.

The name and type of a VDM parameter are mapped to the property `name`, `type` and the `multiplicity`² of the `Parameter` meta-class.

Appendix I

Using VDM Values in Java

As described in Chapter 16 integration between Overture and Java code can be established, either by writing native libraries in Java that can be called from VDM, or by giving a Java program overall control of a VDM model by making calls to that model as a user interacts with a GUI.

In both cases, internal VDM values have to be handled by Java - either because they are passed as arguments to a Java library, and returned as results to VDM, or because they are returned from a VDM model evaluation to a controlling Java program.

This appendix describes the internal class hierarchy used by Overture to represent internal VDM model values, and describes how a Java program can convert these to Java values (int, long, String etc.) as well as creating internal values for returning to the VDM model (e.g. as the return value of library methods).

I.1 The Value Class Hierarchy

All internal VDM values in Overture are held by instances of the `Value` class with the fully qualified name, `org.overture.interpreter.values.Value`. The `Value` class itself is abstract, but subclasses can be instantiated to represent any VDM value, such as a “seq of char”, “nat1” or a value of an arbitrarily complex type. The hierarchy is shown in Figure I.1.

Generally, the name of the `Value` subclass for a VDM type is on the form `<name>Value`, for example `BooleanValue` or `SeqValue`.

The following sections describe how to obtain Java values from a `Value` object, and how to create `Value` objects from basic Java values (or iteratively from other values).

I.2 Primitive Values

Most primitive VDM types have subclasses with simple constructors that take a Java primitive type as an argument:

```
public BooleanValue(boolean value)
public CharacterValue(char value)
```



```
public RealValue(double value) throws Exception
public RationalValue(double value) throws Exception
public IntegerValue(long value)
public NaturalValue(long value) throws Exception
public NaturalOneValue(long value) throws Exception
public QuoteValue(String value)
public NilValue()
```

The constructors that throw exceptions are the ones for which some Java value does not match the VDM type concerned. For example, a `RealValue` or `RationalValue` cannot take the Java values `Double.NaN` or `Double.POSITIVE_INFINITY` as a constructor argument. Similarly, `NaturalValue` and `NaturalOneValue` cannot take a negative long as an argument.

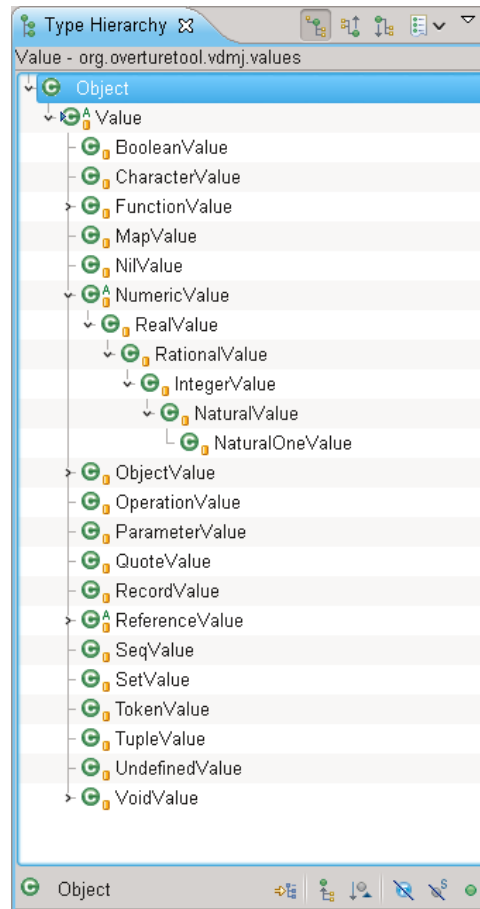


Figure I.1: Java Value Hierarchy

Note that a `QuoteValue` is constructed with a string. This is simply the string value that would appear between angle brackets in VDM, for example `<FAIL>` would be constructed with the Java string `"FAIL"`.



To convert a VDM value into a Java value, the `Value` class provides a number of conversion methods, each of which returns the corresponding Java primitive value, or throws an exception if the conversion cannot be made for the VDM type concerned:

```
public boolean boolValue(Context ctxt) throws ValueException
public char charValue(Context ctxt) throws ValueException
public double realValue(Context ctxt) throws ValueException
public double ratValue(Context ctxt) throws ValueException
public long intValue(Context ctxt) throws ValueException
public long natValue(Context ctxt) throws ValueException
public long nat1Value(Context ctxt) throws ValueException
public String quoteValue(Context ctxt) throws ValueException
```

Note that all of these conversion functions take a `Context` parameter as argument and potentially throw a `ValueException`. The `Context` parameter is used internally by `Overture` and represents the call stack during the evaluation of an expression. This parameter can be set to null when using these methods in Java code outside `Overture`. A `ValueException` is thrown if the VDM value cannot be converted into the Java type requested. For example, calling `booleanValue` on a `RealValue` object will raise a `ValueException` with the message text "Can't get bool value of real".

I.3 Sets, Sequences and Maps

VDM allows primitive types to be built into more complex aggregations and collections, and these can also be converted to and from Java types, though the process is a little more involved. Three classes are provided to assist with this conversion: `ValueSet`, `ValueList` and `ValueMap` (all within the same `org.overture.interpreter.values` package). These classes represent, respectively, a Java `Set`, `List` and `Map` of VDM values:

```
public class ValueSet extends Vector<Value>
public class ValueList extends Vector<Value>
public class ValueMap extends LinkedHashMap<Value, Value>
```

Note that the `ValueSet` class is actually based on a Java `Vector`, not a Java `Set` type, though the class does have set semantics (no duplicates). This is an implementation detail and allows `Overture` to permute set orderings in certain circumstances.

These three classes have obvious constructors, and allow `Values` (or collections of them) to be added to the collection subsequently, using standard Java collection methods:

```
public ValueSet()
public ValueSet(int size)
public ValueSet(ValueSet from)
public ValueSet(Value v)
```



```
public ValueList()
public ValueList(ValueList from)
public ValueList(Value v)
public ValueList(int size)

public ValueMap()
public ValueMap(ValueMap from)
public ValueMap(Value k, Value v)
```

Using these three helper classes, it is now possible to create VDM set, sequence and map values, using constructors of the `SetValue`, `SeqValue` and `MapValue` classes:

```
public SetValue()
public SetValue(ValueSet values)

public SeqValue()
public SeqValue(ValueList values)
public SeqValue(String s)

public MapValue()
public MapValue(ValueMap values)
```

Note that there is a special constructor for `SeqValue` that takes a Java string. This creates a VDM “seq of char”, but without the need to create a `ValueList` with `CharacterValues`.

If the `ValueList` (or another) collection passed to these constructors contains a mixture of VDM types - i.e. a mixture of VDM `Value` subclasses, such as a `BooleanValue` and a `NaturalOneValue` - then the type of the constructed VDM value is the union of the various types passed, in this example “seq of (bool | nat1)”. If this VDM type is not compatible with the use of a corresponding value in the VDM model a dynamic type exception occurs when the value is processed by the model.

Lastly, as before, to get the primitive Java values of a VDM collection, the following methods are provided:

```
public ValueList seqValue(Context ctxt) throws ValueException
public String stringValue(Context ctxt) throws ValueException
public ValueSet setValue(Context ctxt) throws ValueException
public ValueMap mapValue(Context ctxt) throws ValueException
```

Note that, as with the `SeqValue` constructor, there is a special method to return a Java String from a “seq of char” `SeqValue`, rather than a `ValueList` of `CharacterValues`. As before, if the `Value` being used is not a sequence, set or map, then these methods will throw a `ValueException`.



I.4 Other Types

The sections above describe how to create or deconstruct simple VDM values in Java as well as simple collections of these. The remainder of this section describes the unusual cases, for more sophisticated types.

I.4.1 Function values

Overture has an internal `FunctionValue` class used for holding values of functions (e.g. the value of a “lambda” expression or the value of a function defined within a module). But as far as Java is concerned, these values are opaque - there is no equivalent Java construct, and the only way to evaluate a VDM function is to let Overture perform that evaluation. Similarly, Java cannot construct a `FunctionValue`.

The only operation that Java can reasonably perform with a `FunctionValue` is to create a composite function (eg. “`f1 comp f2`” in VDM) or a function iteration (e.g. “`f ** 3`” in VDM) using existing `FunctionValues`. In order to do this, there are two subclasses of `FunctionValue`, called `CompFunctionValue` and `IterFunctionValue`, the constructors for which are as follows:

```
public CompFunctionValue(FunctionValue f1, FunctionValue f2)
public IterFunctionValue(FunctionValue function, long count)
```

These both create new `FunctionValues`, which when evaluated by Overture act as the composition and iteration of the arguments, respectively.

There is a method for obtaining a `FunctionValue` from a `Value`, but note that this is not an internal Java value (unlike other `Value` methods, like `realValue`). It is used as a more convenient way of casting the `Value` to a `FunctionValue`.

```
public FunctionValue functionValue(Context ctxt)
```

I.4.2 Object Values

When VDM++ and VDM-RT create new objects using the “new” operator, the resulting values are held as `ObjectValues` in Overture. These are complex types that involve function and operation definitions for the object as well as any type, value, sync, thread or traces sections defined. Therefore `ObjectValues` are really opaque to Java and cannot be used directly.

Like for `FunctionValue`, the `ObjectValue` class has a method for converting a `Value` into an `ObjectValue`:

```
public ObjectValue objectValue(Context ctxt)
```



I.4.3 Record Values

A VDM record is just a collection of typed field values. A `RecordValue` can be obtained from a `Value` using the following method, which returns a `RecordValue` rather than some other Java representation:

```
public RecordValue recordValue(Context ctxt)
```

To get individual field values from a `RecordValue`, two more Java helper types have to be introduced, called `FieldMap` and `FieldValue`. A `FieldValue` has the following constructor, and represents a record field:

```
public FieldValue(String name, Value value, boolean comparable)
```

The `comparable` argument indicates whether this field is used in the value comparison between record values. A field declared with “-” in VDM would have a false argument, but normally this argument would be true, and the value must match the record type being used. `FieldValues` are added to a `FieldMap`, which is just a Java `List` of `FieldValues`.

So given a `RecordValue`, its `FieldMap` can be obtained from a public final field in the object, called `fieldMap`¹, and from there, individual `FieldValues` can be accessed - e.g. `fieldMap.get(0).name` and `fieldmap.get(0).value`.

To create a `RecordValue`, the record type is obtained from the `RemoteInterpreter`:

```
type = remoteInterpreter.getInterpreter().findType(typename)
```

The type is then passed to the `RecordValue` constructor, along with a `FieldMap` or a `ValueList` (of the fields in order).

```
public RecordValue(RecordType type,  
                  ValueList values,  
                  Context ctxt)  
public RecordValue(RecordType type,  
                  FieldMap mapvalues,  
                  Context ctxt)
```

The `Context` parameter is needed to allow records with invariants to check the invariant before the value is constructed. Note that currently, record types with an invariant cannot be constructed in Java. The `Context` parameter can be passed as null from Java.

The caller is responsible for passing field values that match their expected type. If they do not match, Overture throws a dynamic type exception for subsequent evaluations.

I.4.4 Token Values

Token values are simply wrappers for normal VDM values, reflecting the way they are created in VDM, like `mk_token("hello")`, which would be a wrapper for a “seq of char”. There is no

¹Really this ought to have a `get` method.



special way of getting a `TokenValue` from a `Value`, other than casting it. Having casted the `Value`, the wrapped value can be obtained from the public final `Value` field called “value”.

Constructing a `TokenValue` is just a matter of passing the `Value` required:

```
public TokenValue(Value exp)
```

I.4.5 Tuple Values

A `TupleValue` in `Overture` is a wrapper for a `ValueList`. The following method and constructor can be used like one would expect:

```
public TupleValue(ValueList argvals)
public ValueList tupleValue(Context ctxt)
```

I.4.6 Invariant Values

A VDM type can be given a name and an invariant, e.g. when wrapping a primitive type without an invariant. `Overture` has a separate `Value` subclass for values of such types that simply combine the primitive `Value` with a `FunctionValue` for the invariant. However, as with `RecordValues` (which can also have invariants), it is not currently possible to create `InvariantValues` in Java for types that have an invariant.

For types without an invariant, the constructor is as follows:

```
public InvariantValue(NamedType type, Value value, Context ctxt)
```

The `NamedType` is obtained in a similar way to the `RecordType` above, using the `RemoteInterpreter`. Note that the caller is responsible for passing a `Value` that matches the expected type. If they do not match, `Overture` will throw a dynamic type exception for subsequent evaluations.

I.4.7 Void Values

Operations which do not return a value in VDM (i.e. `==> ()`) return an instance of `VoidValue` in Java. The constructor has no arguments.