# User Guide for the Overture VDM Tool Support

by

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

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

**Document history**

| Month | Year | version of Overture.exe |
|---------|------|-------------------------|
| January | 2010 | 0.1.5 |
| March | 2010 | 0.2 |

# Contents

**ABSTRACT**

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

# 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 [**?**, **?**, **?**]. 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 the model using Overture help to identify areas of incompleteness or ambiguity in informal system specifications, and provide some level of confidence that a valid implementation will have key properties, especially those of safety or security. VDM has a strong record of industrial application, in many cases by practitioners who are not specialists in the underlying formalism or logic [**?**,**?**,**?**]. Experience with the method suggests that the effort expended on formal modeling and analysis can be recovered in reduced rework costs arising from design errors.

VDM models can be expressed in a specification language (VDM-SL) that supports the description of data and functionality [**?**, **?**, **?**]. 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 the user to add any relevant constraints as data type invariants. Functionality is defined in terms of operations over these data types. Operations can be defined implicitly by preconditions and postconditions that characterize their behavior, or explicitly by means of specific algorithms. An extension of VDM-SL, called VDM++, supports object-oriented structuring of models and permits direct modeling of concurrency [**?**]. An additional extension to VDM++ is called VDM Real Time (VDM-RT) (formerly called VDM In a Constrained Environment (VICE)) [**?**, **?**]. All these different dialects are supported by the unified tool called Overture.

Since the VDM modeling languages have a formal mathematical semantics, a wide range of analyses can be performed on models, both to check internal consistency and to confirm that models have emergent properties. Analyses may be performed by inspection, static analysis, testing or mathematical proof. To assist in this process, Overture supply tool support for building models in collaboration with other modeling tools, to execute and test models and to carry out different forms of static analysis [**?**]. It can be seen as an open source version of the commercial tool called VDM-Tools [**?**, **?**, **?**] although features to generate executable code in high-level programming languages are not yet available in Overture.

This guide explains how to use the Overture IDE for developing models for different VDM dialects. This user manual starts with an explanation of how to get hold of the software in Section 2. This is followed in Section 3 with an introduction to the concepts used in the different Overture perspectives based on Eclipse terminology. In Section 4 it is explained how projects are managed in the Overture IDE. In Section 5 the features supported when editing VDM models are explained. This is followed in Section 6 with an explanation of the interpretation and debugging capabilities in the Overture IDE. Section 7 then illustates how test coverage information can be gathered when models are interpreted. Afterwards, Section 8 shows how models with and without test coverage information can be generated to the text processing system LaTeX and automatically converted to `pdf` format if one have `pdflatex` installed on the computer. Afterwards, from Section 9 to Section 13 different VDM specific features are explained. In Section 9 the use of the notion for

proof obligations and its support in Overture is explained. In Section 10 a notion of combinatorial testing and the automation support for that in Overture is presented. In Section 11 support for mapping between object-oriented VDM models and UML models is presented. In Section 12 it is illustrated how a VDM++ project can be changed into a new VDM-RT project. In Section 13 it is shown how to analyse and display logs from executing such VDM-RT models. After these sections the main part of the user manual is completed in Section 16 with an explanation of the features from Overture which are also available from a command-line interface. Appendix A provide a list of all the standard templates built into Overture. From Appendix B to G complete lists of different kinds of errors, warnings and proof obligations are provided and further explanation are provided where it is judged necessary. Finally, in Appendix I an index of significant terms used in this user manual can be found.

## 2 Getting Hold of the Software

Overture is an open source tool, developed by a community of volunteers and built on top of the Eclipse platform. The project to develop the tools is managed using SourceForge. 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://sourceforge.net/projects/overture`

you can use the *Download Now* button to automatically download a pre-installed versions of Overture for your operating system. Supported systems are: Windows, Linux and Mac[1]. Note that when you have extracted all files from the zip file with the Overture executable for your selected operating system you will find the first time you start it up it will ask you for selecting a workspace. Here we simply recommend you to chose the default one proposed by Overture and tick off 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.

Large libraries of sample VDM-SL, VDM++ and VDM-RT models is available and can be downloaded from SourceForge under the `files/Examples` section using the URL[2]:

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

Such existing projects can be imported as described in subsection 4.1.

---

[1]It is planned to develop an update facility, allowing updates to be applied directly from within the generic Eclipse platform 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.

[2]The library files are created to be used with Eclipse, but can be opened with file compression programs like `Winrar` on Windows

# 3   Using the Overture Perspective

## 3.1   Getting into the 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. Thus, if a user is familiar with one Eclipse product, it will generally be easy to start using a different product on the same workbench. The Eclipse workbench consists of several panels known as *views*, such as the Script Explorer view at the top left of Figure 1. A collection of panels is called a *perspective*, for example Figure 1 shows the standard Overture 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.
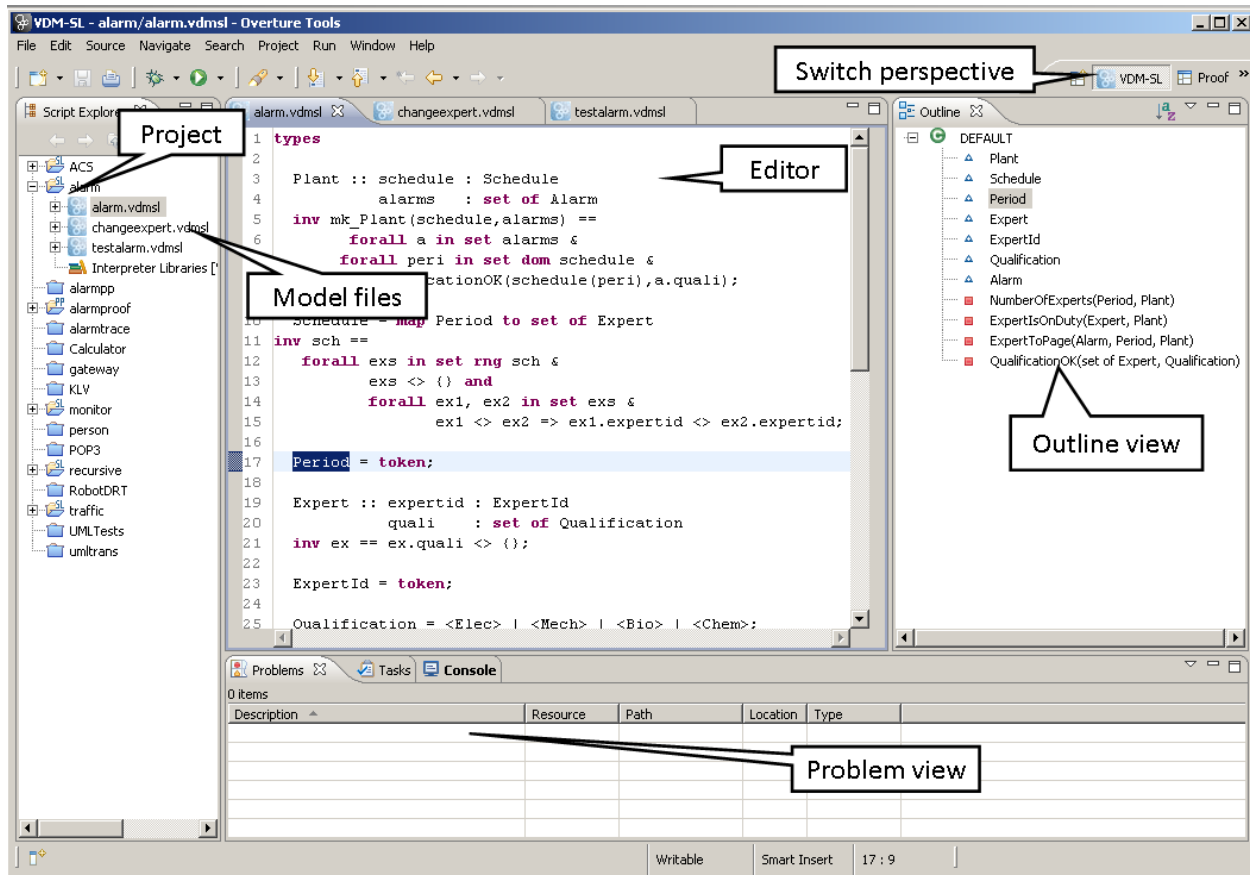


Figure 1: The Overture Perspective

The *Script 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.

Depending upon the dialect of VDM used in a given project, a corresponding Overture editor will be available here. A new VDM project is created choosing the *File → New → Project* resulting in Figure 2. Here select the desired VDM dialect and press *Next*. Finally, a name needs to be given to the project and the simply select `Finish`.

Figure 2: Creating a New VDM Project
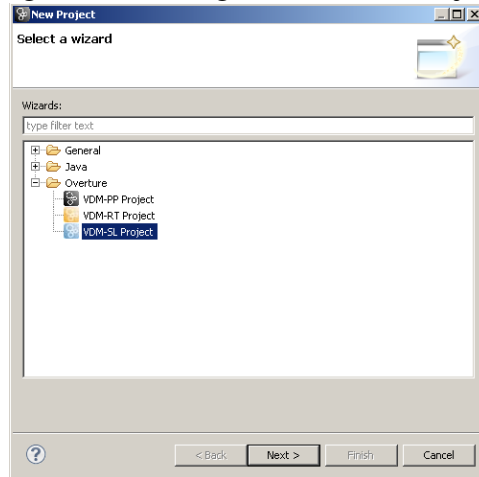


The *Outline view*, on the right hand side of Figure 1, presents an outline of the file selected in the editor. The outline displays any declared VDM definitions such as their state components, values, types, functions and operations. In case of a flat VDM-SL model the module is called `DEFAULT`. Figure 3 shows a closeup of the outline view. Clicking on an operation or function will move the cursor in the editor to the definition of the operation. At the top of the outline view there is a button to sort what is displayed in the outline view, for instance it is possible to hide variables.
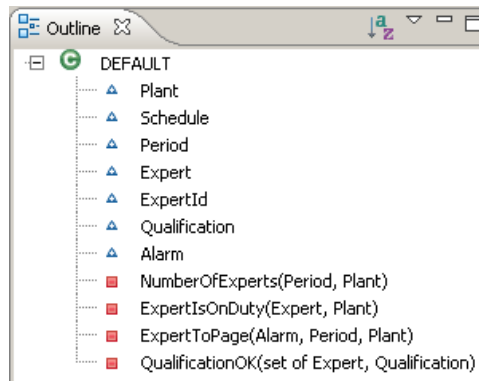


Figure 3: The Outline View

The *Problems view* at the buttom of Figure 1 gathers information messages about the projects you are working on. This includes information generated by Overture, such as warnings and errors.

Please use the suggested corrections with caution, since the tool cannot guess what you intended to write.

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. One convenient feature is a toolbar of shortcuts to switch between different perspectives that appears on the right side of the screen; these vary dynamically according to context and history.

## 3.2 Additional Eclipse Features Applicable in Overture

### 3.2.1 Opening and Closing Projects

In order not to take up too much space and computing power it may be advantageously to close projects that are not used currently. This can be done by right clicking such projects and then selecting the *Close Project* entry in the For such closed projects it is possible to open them again in the same fashion using the *Open project* entry in the same right click menu.

### 3.2.2 Adding Additional VDM File Extensions

If one would like to use additional file types to be associated with a particular VDM editor instead of the standard `vdmsl`, `vdmpp` and `vdmrt` file types this is possible in Overture. This is done using the *Window → Preferences* menu point. Here one can start typing `content types` which will result in a menu similar to Figure 4. Here one can press the `Add` button for the appropriate content type that one wishes to add additional types of file extensions.

### 3.2.3 Remove Directories without Source Files

In case the directory with the VDM source files includes subdirectories without source files it may be comvenient to remove this additional noice from the *Explorer* view. This can be acomplished by pressing the small downward pointing arrow at the top right-most corner of the *Explorer* view. Then the menu shown in Figure 5 will pop up and here *Filters...* is selected and then different parts can be filtered away including directories that have no source files.

### 3.2.4 Including line numbers in the Editor

In case line numbers are desired in the Overture editor it is possible to to right click in the left-hand-side margin of the editor and then select `show line numbers` as shown in Figure 6.

## 4 Managing Overture Projects

## 4.1 Importing Overture VDM Projects

It is possible to import Overture VDM projects by right-clicking the explorer view and selecting *Import*, followed by *General → Existing Projects into Workspace*. In this way the projects from
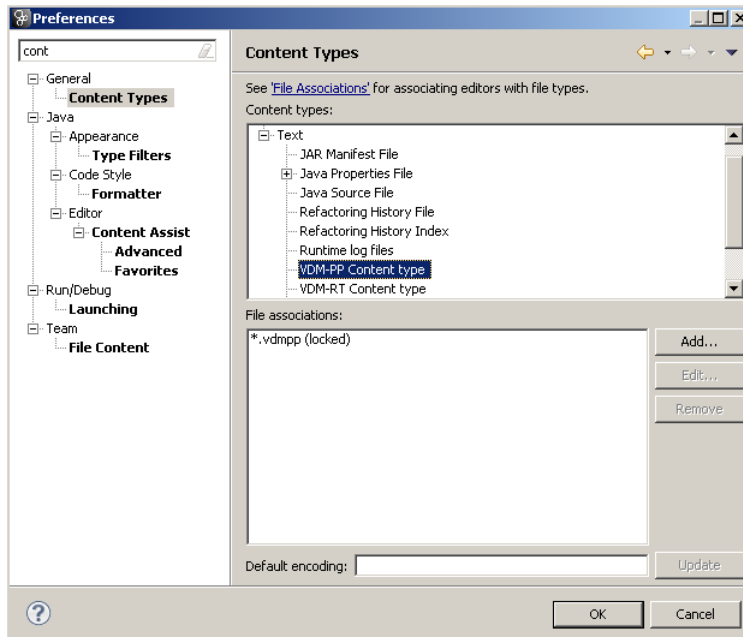
Figure 4: Adding Additional Contents Types
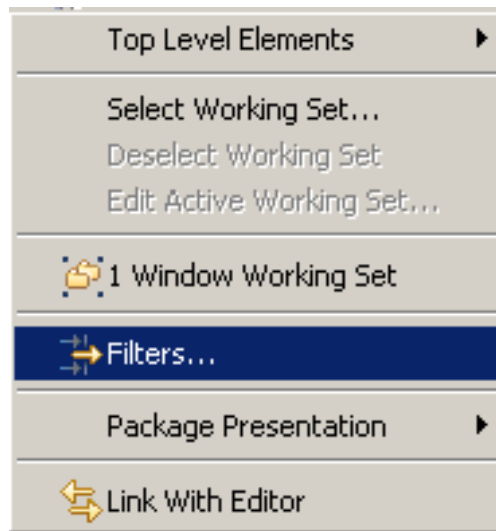


Figure 5: Filtering Directories without source files

`.zip` files mentioned in Section 2 can be imported very easily.
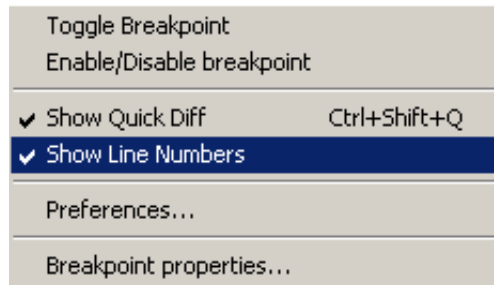
Figure 6: Adding Line Numbers in Editor

## 4.2 Creating 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. Press *Next*

4. Type in a project name

5. Chose whether you would like the contents of the new project to be in your workspace or outside from existing source files (browse to the appropriate directory) and

6. click the finish button (see Figure 7).

## 4.3 Creating Files

Switching to the Overture perspective will change the layout of the user interface to focus on the VDM development. To change perspective go to the menu window → open perspective → other... and choose the Overture perspective. When the developer is in the Overture Perspective the user 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 one needs to choose a file name and optionally choose a directory if one does not want to place the file in the directory of the chosen Overture project. Then a new file with the appropriate file extension according to the chosen dialect (vdmsl, vdmpp or vdmrt) can be created in the selected directory. This file will use the appropriate module/class template to get the user started with defining the module/class meant to be placed in this new file. Naturally, keywords not used can be deleted.
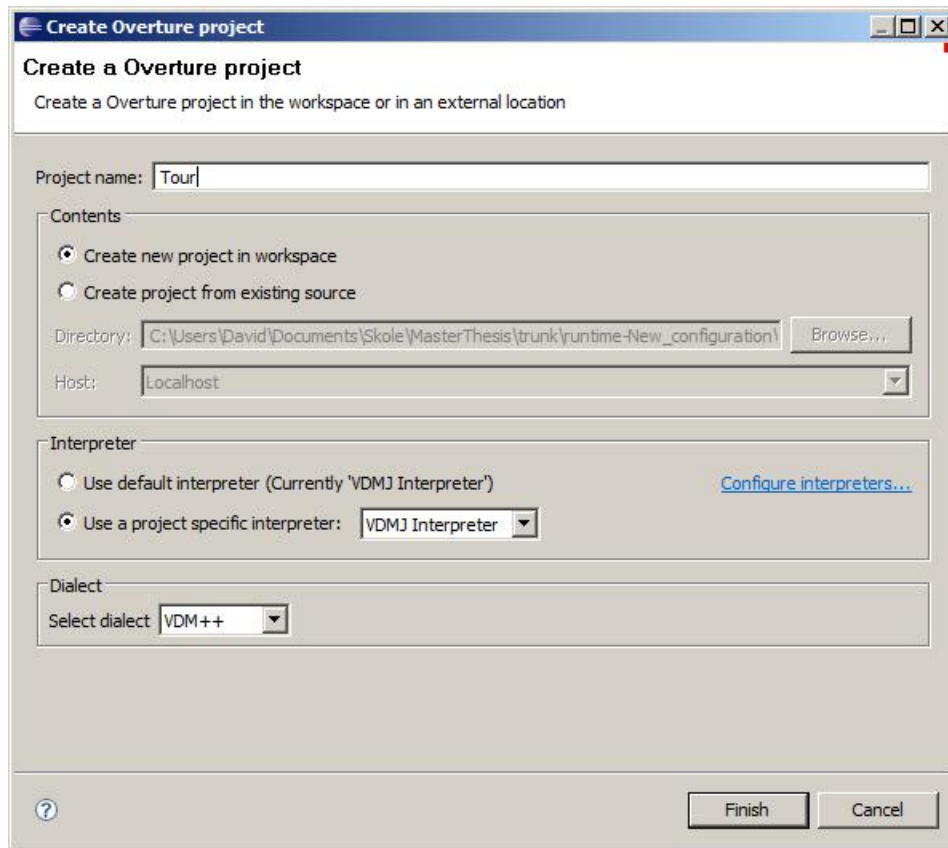
Figure 7: Create Project Wizard

## 4.4   Setting Project Options

For each Overture VDM project it is possible to set various VDM specific settings. One can get access to these by selecting a project in the *Explorer view* and then right clicking and selecting *properties*. Here, a VDM specific settings property which looks like in Figure 8 is shown. The options that can be set for each VDM project are:

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

**Suppress type checking warnings:** This flag is per default not set but if one would like to switch off such warning the flag can be set here.

**Dynamic type checks:** This is an option to the interpreter (default on) for continuously type checking the values during interpretation of a VDM model. It is possible to swich off the check here.
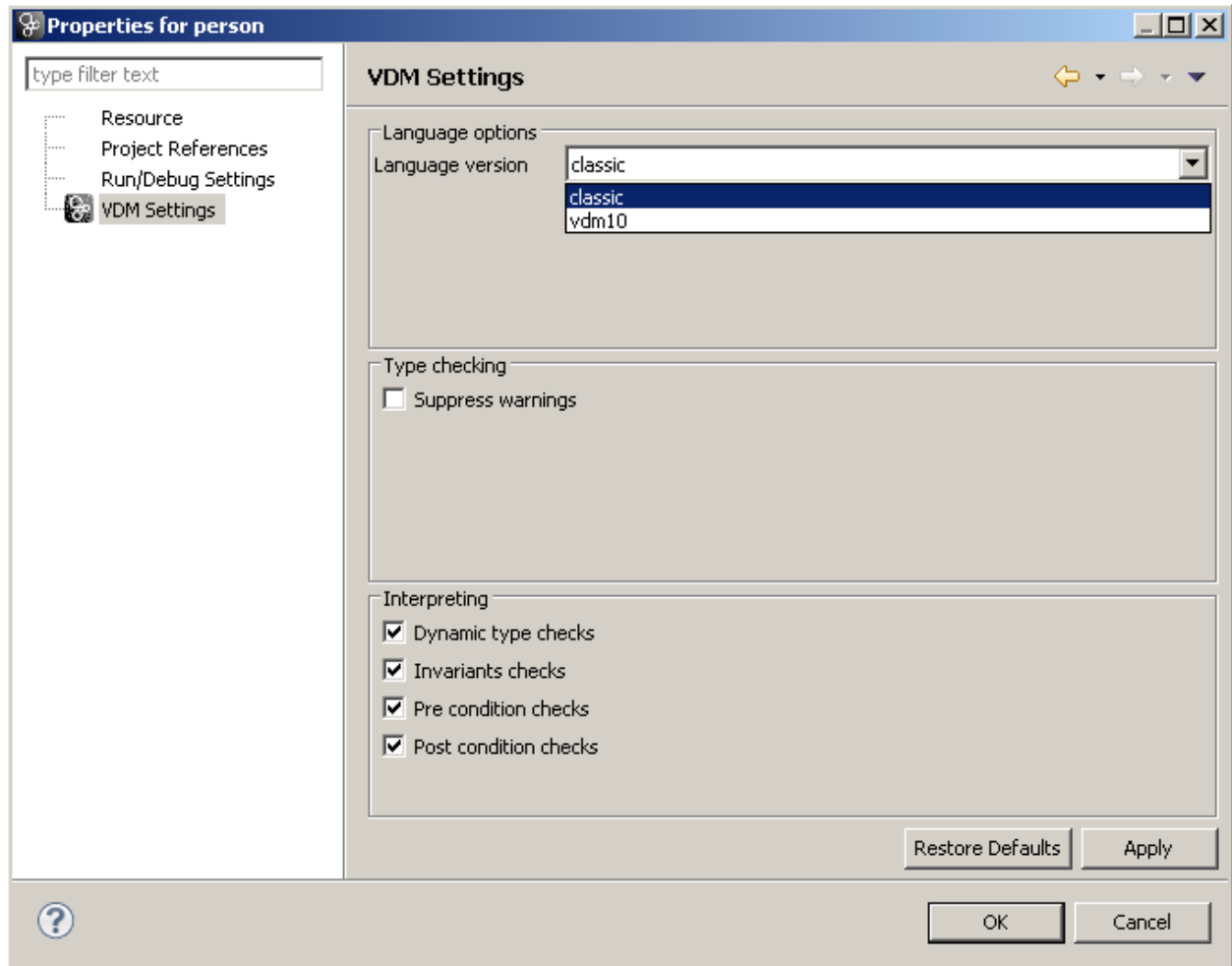
Figure 8: Overture Project Settings

**Invariant checks:** This is an option to the interpreter (default on) for continuously also checking both state and type invariants of the values during interpretation of a VDM model. It is possible to swich off this check here but note that option requires dynamic type checking also to be swiched on.

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

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

In addition to these project settings it is possible to chose to have a main LaTeX file where one can incorporate the explanations outside the `vdm_al` environment and decide the order in which the different VDM source files shall be included. TBW

# 5 Editing VDM models

## 5.1 VDM Dialect Perspectives

Whenever one wishes to edit parts of a VDM model it can be done in the editor view. In general it is recommended to make use of the VDM dialect perspective when one wish to carry out the editing, since browsing in the VDM model is supported both by the editor view as well as by the Outline view. Whenever editing is carried out in the edit view syntax checking is carried out continuously (even before the files are saved). Once files are saved the syntax checking is accompanied by type checking of the entire VDM model if no syntax errors are found. As a result new problems (errors and/or warnings) can be found. These will be displayed both in the problems view as well as with small icons in the editor view at the lines where the problems have been identified.

## 5.2 Using Templates

Templates can be particularly useful when modifying VDM models of all three dialects (SL, PP and RT). If you hit the key combination *CTRL+space* after the initial characters of the template needed, Overture triggers a proposal. For example, if you type "fun" followed by *CTRL+space*, the Overture IDE will propose the use of an implicit or explicit function template as shown in Figure 9. The Overture IDE supports several types of template: cases, quantifications, functions (explicit/implicit), operations (explicit/implicit) and many more. Additional templates can easily be added in the future. The use of templates makes it much easier for users lacking deep familiarity with VDM syntax to nevertheless construct models.

It is possible to adjust and add to the templates enabled inside the Overture editor. This can be done by selecting the menu called *Window → Preferences*. By starting to type *template* the Overture specific preferences for templates will become visible. In Figure 10 it can be seen how the template for cases expressions is defined in Overture. Note that new templates can be added and the already defined ones can be adjusted by selecting the *Edit* button. It is also possible to remove templates using the *Remove* button. A full list of the standard defined templates is available in Appendix A.

# 6 Interpretation and Debugging in Overture

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

```
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 9: Explicit function template

## 6.1 Debug configuration

Debugging the model under development is done by creating a debug configuration from the menu *Run → Debug configuration . . .* The debug configuration dialog requires the following information as input to start the debugger: the project name, the class and the starting operation/function. Figure 11 shows a debug configuration. Clicking one of the browse buttons will open a dialog which give the user a list of choices. The class and operation/function are chosen from the dialog with the list of expandable classes, if the operation or function have arguments these must be typed in manually. Alternatively one can get to the *Debug Configuration* by right clicking on a project in the Explorer view and then selecting the *Debug As → debug Configuration*. Finally it is also possible to get to the *Debug Configuration* by using the small downwards pointing arrow next to the debug icon ( ) at the top of the Overture tool.

## 6.2 Debug Perspective

The Debug Perspective contains the views needed for debugging in VDM. Breakpoints can easily be set at desired places in the model, by double clicking in the left margin. When the debugger reaches the location of the breakpoint, the user can inspect the values of different identifiers and step through the VDM model line by line.

The debug perspective shows the VDM model in an editor as the one used in the Overture Perspective, but in this perspective there are also views useful during debugging. The features
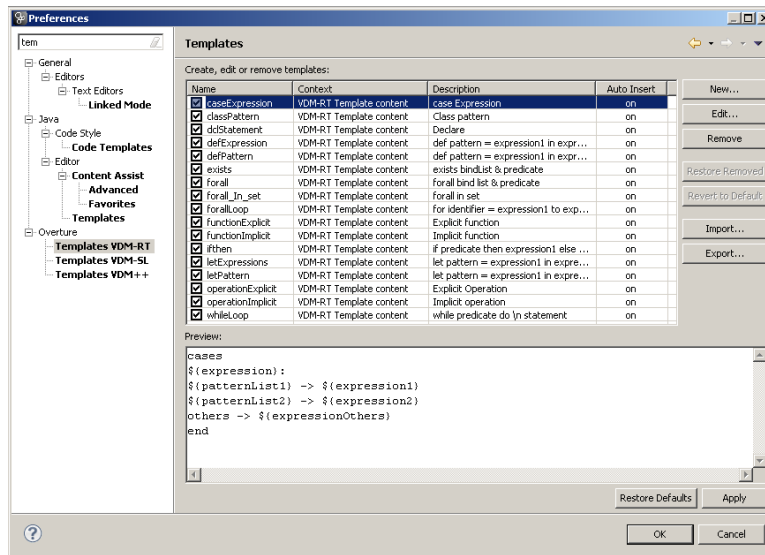
Figure 10: Adjusting templates for Overture
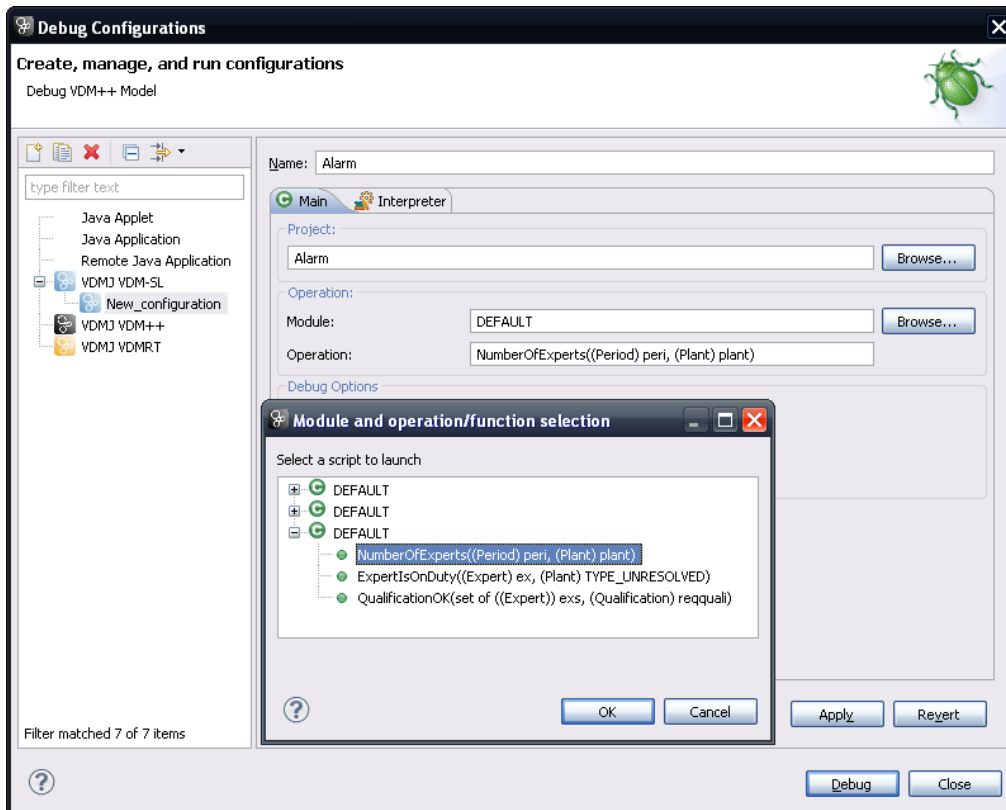


Figure 11: The debug configuration dialog

Table 1: Overture debugging buttons

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

provided in the debug perspective are described below. The Debug Perspective is illustrated on Figure 12
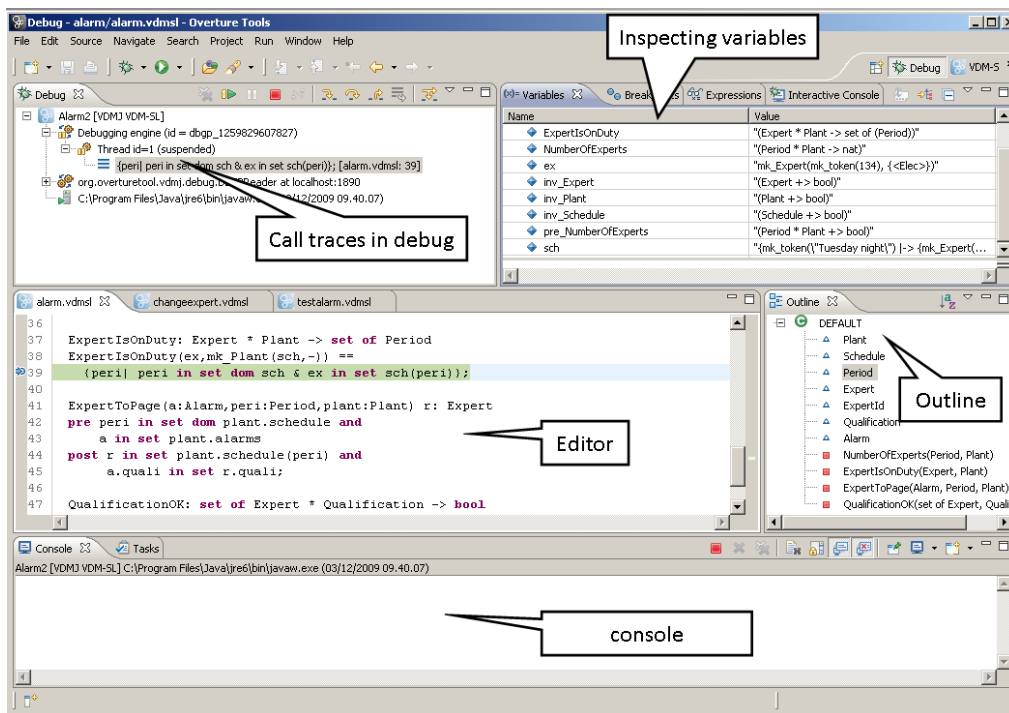


Figure 12: Debugging perspective

### 6.2.1   Debug View

The debug View is located in the upper left corner in the Debug Perspective - see Figure 12. The debug view shows all running models and the call stack belonging to them. It also displays whether a given model is stopped, suspended or running. In the top of the view buttons for debugging such as; stop, step into, step over, resume, etc. are located. All threads are also shown, along with their running status. It is possible to switch between threads from the Debug View.

At the top of the view are buttons for controlling debugging such as; stop, step into, step over and resume. These are standard Eclipse debugging buttons (see Table 1).

### 6.2.2   Variables View

This view shows all the variables in a given context, when a breakpoint is reached. The variables and their values displayed are automatically updated when stepping through a model. The variables view is by default located in the upper right hand corner in the Debug Perspective. It is also possible to inspect complex variables, expanding nested structures and so forth.

### 6.2.3   Breakpoints View

Breakpoints can be added both from the edit perspective and the debug perspective from the editor view. In the debug perspective however, there is a breakpoints view that shows all breakpoints. From the breakpoints view the user can easily navigate to the location of a given breakpoint, disable, delete or set the hit count or a break condition. In Figure 12 the Breakpoints View is hidden behind the Variables View in the upper right hand corner in a tabbed notebook.

### 6.2.4   Conditional breakpoints

Conditional breakpoints can also be defined. These are a powerful tool for the developer since it allows specifying a condition for one or more variables which has to be true in order for the debugger to stop at the given breakpoint. Apart from specifying a break condition depending on variables, a hit count can also be defined. A conditional breakpoint with a hit count lets the user specify a given number of calls to a particular place at which the debugger should break.

Making a breakpoint conditional is done by right clicking on the breakpoint mark in the left margin and select the option Breakpoint properties. . . This opens a dialog like the one shown in Figure 13. It is possible to choose between two different conditional breakpoints, a hit count condition and one based on an expression defined by the user.

### 6.2.5   Expressions View

The expressions view allows the user to write expressions used for inspecting values of the model. As for the variables view, the expressions are automatically updated when stepping throught the model. Watch expressions can be added manually or created by selecting 'create watch expression' from the variables view. It is of course possible to edit existing expressions. Like the Breakpoints View this view is hidden in the upper right hand corner.
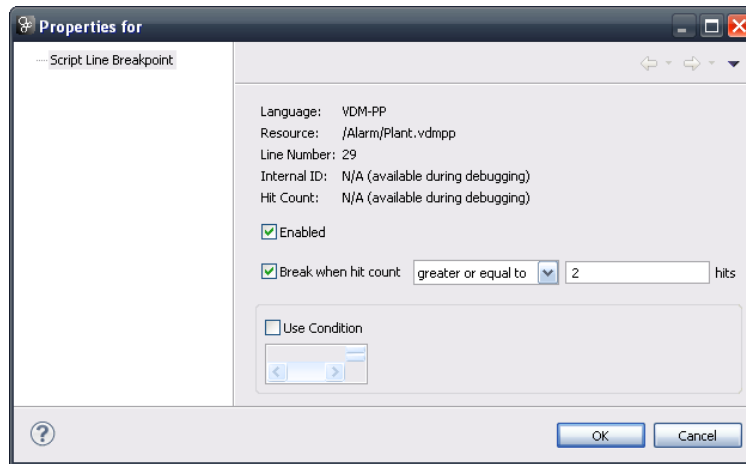
Figure 13: Conditional breakpoint options

### 6.2.6 Interactive Console View

While the Expressions View allows to easily inspect values, the functionality is somewhat limited compared to the functionality provided by VDMTools. For more thorough inspections the Interactive Console View is more suited. Here commands can be executed on the given context, i.e. where the debugger is at a breakpoint. The Interactive console keeps a command history, so that already executed commands can be run again without actually typing in the command all over. Figure 14 shows the interactive console.
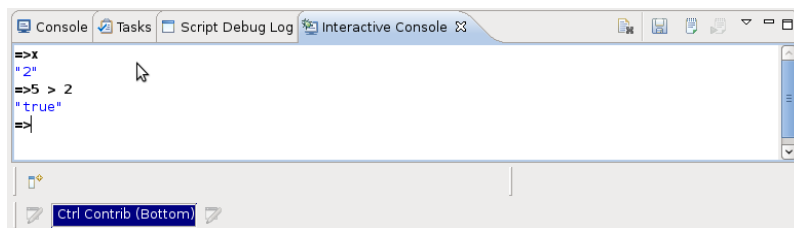


Figure 14: The interactive console

## 7 Collecting Test Coverage Information

When a VDM model is being interpreted it is possible to automatically collect test coverage information. Test coverage measurement helps you to see how well a given test suite covers the VDM model. This is done by collecting information in a special test coverage file about which statements and expressions are evaluated during the execution of the test suite.

More

# 8  Pretty Printing to LaTeX

Include `overture.tex` which among other things makes use of the `times.cls` and `listings.cls` style classes. This enables the use of the standard `lstlisting` environment for type setting source text and display it in a tele-type proportional font where all VDM keyword are typeset in a bold font. Per default the listings will be inserted into boxes but it is easy to adjust (using the parameters to the `lstlisting` environment) if no boxes are desired.

It is possible to use literate programming/specification [**?**] just as inside VDMTools. Then one needs to use the LaTeX text processing system with plain VDM models mixed with textual documentation. The VDM model parts must be enclosed within "`\begin{vdm_al}`" and "`\end{vdm_al}`". The text-parts outside the specification blocks are ignored by the parser (but used by the pretty-printer).

# 9  Managing Proof Obligations

In the different VDM dialects it is possible to identify places where run-time errors potentially could occur if the model was to be executed. In essence these can be considered as additional to the existing type checking performed. Just like almost all other computer based languages it is not possible to automatically statically check if such places indeed could result in a run-time error or not. Thus Overture provides socalled "proof obligations" for all places where such run-time errors "could" occur. Each *Proof Obligation* (PO) is formulated as a predicate that must hold on a particular place of the VDM model and thus it may have particular context information associated with it. These POs can be considered as constraints that will gurantee the internal integrity of the VDM models if they are all correct. In the long term it will be possible to prove these constraints by a proof component in Overture but this is not yet working as well as we wish.

It takes a little time for newcommers to VDM to get used to the form of these so it may be worthwhile to elaborate a bit on the form of the proof obligations. Theses can be divided into different categories depending upon their nature. These can be found in Appendix G along with a short explanation 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 a selected VDM file one can right click in the *Explorer* view and then select the *Proof Obligations → Generate Proof Obligations* menu item. Overture will then change into a special *Proof Obligations Perspective* as shown in Figure 15.

Note that in the *Proof Obligation Explorer* view each proof obligation have a number of components:

- A unique number in the list shown;

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

- The proof obligation category (type) and finally

- A status field indicating whether the proof obligation is trivially correct or would have to be proved by a normal proof engine.
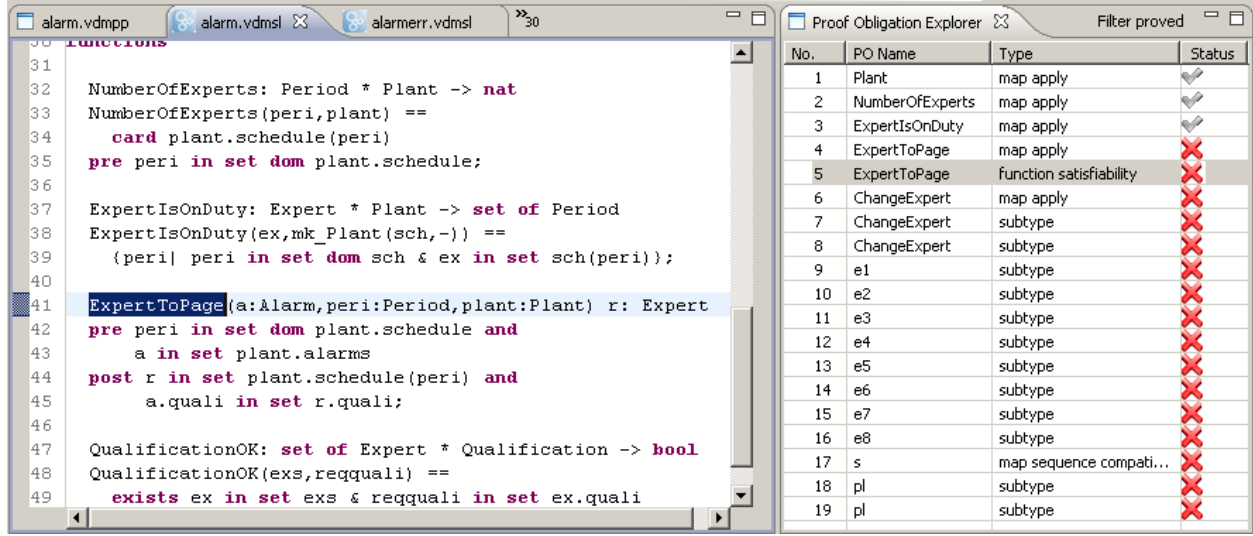
Figure 15: The Proof Obligation perspective

Note that at the top of the *Proof Obligation Explorer* it is possible to filter away all the proof obligations that are trivially correct pressing the *Filter proved* button at the top of this view.

# 10    Combinatorial Testing

In order to automate parts of the testing process a notion of *traces* have been introduced into VDM++ (note that this is only available for VDM-SL models if VDM-10 have been selected). Such traces conceptually correspond to regular expressions that can be expanded to a collection of test cases. Each such test case is then composed as a sequence of operation calls. If a user defines such traces it is possible to make use of a special combinatorial testing perspective that enables the automatic unfolding of the traces and automatic execution of each of the test cases. Subsequently, the results of running all these can be inspected and erroneus test cases can easily be found. The user can then fix the problem and reuse the same traces definitions.

## 10.1    The Use of the Trace Definition Syntax

The syntax for trace definitions are defined as:

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

named trace  =  identifier, { '/', identifier }, ':', trace definition list ;

The naming of trace definitions (with the "/" separator) is used for indicating the paths that are used for generated argument files for test cases (.arg) and the corresponding result files (.res)[3].

---

[3]Currently the full path names are however not supported in an Overture context but this is envisaged in the future.

trace definition list  =  trace definition term, { ‘;’, trace definition term } ;

So the “;” operator is used for indicating a sequencing relationship between its *trace definition term*’s.

trace definition term  =  trace definition
          | trace definition term, ‘|’, trace definition ;

So the “|” operator is used for indicating alternative choices between trace definitions.

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

Trace definitions can have different forms and combinations:

- Core definitions which includes application of operations and bracketed trace expressions.

- Trace bindings where identifiers can be bound to values and in case of looseness (**let** *bind* **in set** *setexpr* **in** *expr*) this will give raise to multiple test cases generated.

- Trace repeat patterns which are used whenever repetition is desired.

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

trace apply expression  =  identifier, ‘.’, identifier, ‘(’, expression list, ‘)’ ;

Trace apply expressions are the most basic element in trace definitions. The identifier before the “.” indicate an object for which the operation (listed after the “.”) is to be applied with a list of arguments (the expression list inside the brackets). Note that with the current syntax for trace definitions, apply expressions are limited to this form `instid.opid(args)` so it is for example not at the moment possible to call an operation in the same class directly as `opid(args)`. Nor is it possible with the current syntax to make use of a particular operation in a superclass in case of multiple possible ones which in VDM++ is would be written as `instid.clid‘opid(args)`. In the current version it is also not allowed to call functions here directly, although that may be changed at some stage in the future.

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

The different kinds of repeat patterns have the following meanings:

- '∗' means 0 to n occurrences (n is tool specific).

- '+' means 1 to n occurrences (n is tool specific).

- '?' means 0 or 1 occurrences.

- '{', n, '}' means n occurrences.

- '{', n, ',' m '}' means between n and m occurrences.

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

  trace bindings  =  trace binding, { trace binding } ;

  trace binding  =  'let',  local definitions, { ',',  local definition }, 'in'
           |   'let',  bind, 'in'
           |   'let',  bind, 'be', 'st',  expression, 'in' ;

## 10.2   Using the Combinatorial Testing GUI

If one have used the **traces** syntax described above it is possible to go to the *Combinatorial testing* perspective. An example of using that perspective can be seen in Figure 16.

Different icons are used to illustrate 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 test cases result in a run-time error other test cases with the same prefix will be filtered away and thereby skipped by in the test execution. The number of skipped test cases is indicated after number of test cases for the trace definition name.

# 11   Mapping VDM++ back and forth to UML

For VDM++ and VDM-RT projects it is possible to move automatically back and forth between a VDM model and its corresponding UML model. Essentially these can be considered as different views of the same model. The UML model is typically used as a graphical overview of the model using class diagrams and the sequence diagrams can be used to indicate the desired test scenarios that a user would like to perform. The VDM++ model is typically used as the model where the
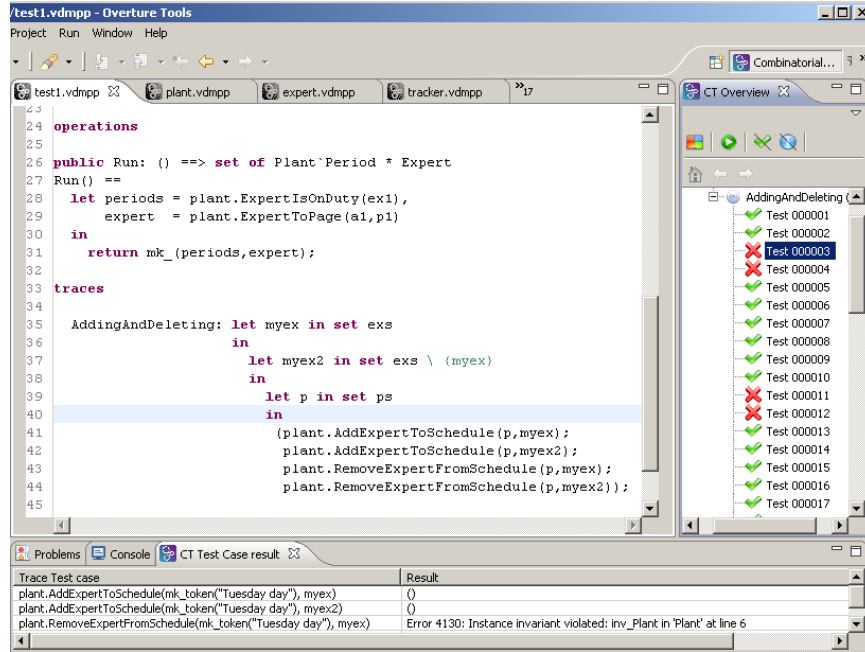
Figure 16: Using Combinatorial Testing

details for each definition can be found and used for detailed semantic analysis. The exchange between VDM++ and UML is done using the XML formal called XMI. At the moment only the UML tool Enterprise Architect is supported. Export from EA is done by selcted the *Project* menu and selecting *Import/Export → Export Package to XMI*. This is illustrated in Figure 17.

Mapping back and forth between a VDM++ model and a UML model is in practice done from the *Explorer* view where right-clicking the project will result in a menu popping up. In this menu there is an entry for *UML transformation*. If this is selected it is either possible to *Import XMI* if one wish to import UML definitions from UML or to *Export XMI* if one wish to go from VDM++ to UML.

At the class diagram at the UML level additional classes will be generated for standard VDM++ basic types. When such UML models are mapped back to VDM++ such additional classes are ignored.

## 12  Moving from VDM++ to VDM-RT

In the methodology for the development of distributed real-time embedded systems using the VDM technology a step is described where one moves from a VDM++ model to a VDM-RT model [**?**]. This step is supported by the Overture tool suite where it is possible to copy a VDM++ project into the starting point for a VDM-RT project. This is done by right clicking on the VDM++ project to be converted in this fashion in the Project Explorer view followed by selecting the *Overture Utility*
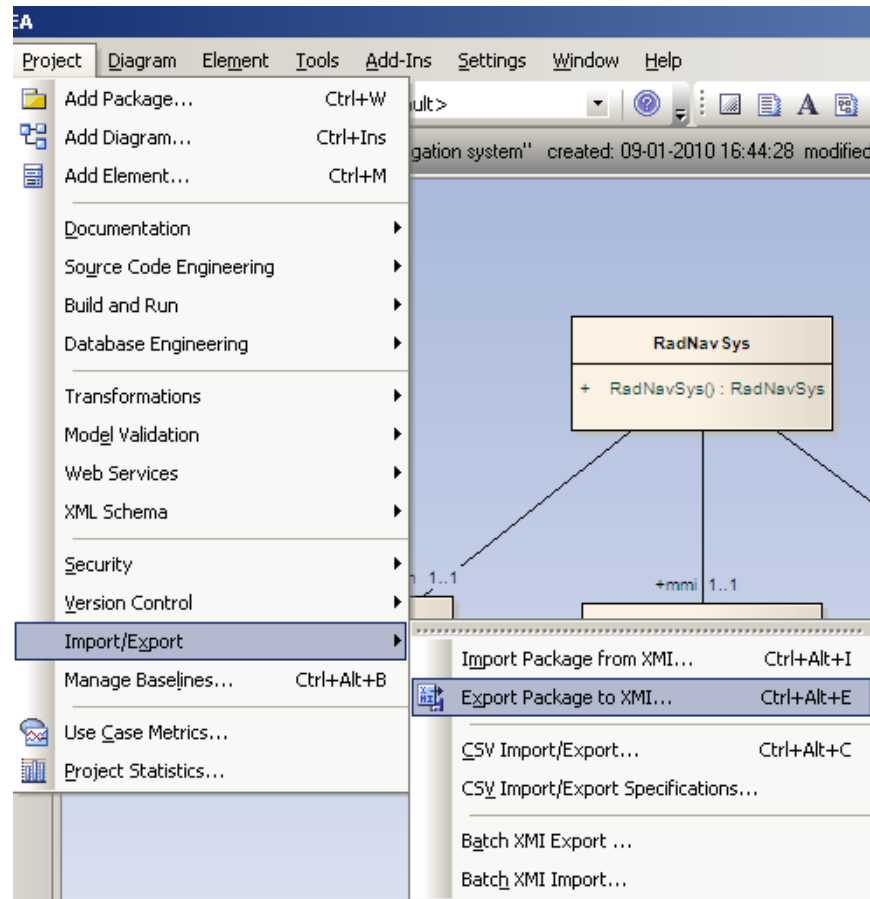
Figure 17: Exporting UML definitions from EA

→ *Create Real Time Project*. As a consequence a new VDM-RT project is created. It will be called exactly the same as the VDM++ project with `RT` appended to the project name. Inside the project all the `vdmpp` files will instead have the `vdmrt` extension. The original VDM++ project is not changed at all. Thus this is simply a quick and easy way to get to the starting point for a VDM-RT model. One then manually need to create a **system** with appropriate declarations of `CPU`s and `BUS`ses.

## 13   Analysing and Displaying Logs from VDM-RT Executions

When a VDM-RT model is being executed a textual logfile is created in a ”`logs/debugconfig`” folder with the *.logrt* extension. The file name for the logfile indicates the time at which it has been written so it is possible to store multiple of these. This logfile can be viewed in the build-in Real-Time Log Viewer, by double-clicking the file in the project view. The viewer enables the user to explore system execution in various perspectives. In Figure 18 the architectural overview of the

system is given, describing the distributed nature of the model.
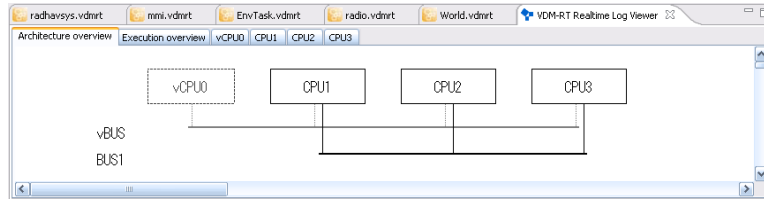


Figure 18: Architectural overview

The RealTime Log Viewer also enables the user to get an overview of the model execution on a system level – this can be seen in Figure 19. This view shows how the different CPUs communicate via the BUSes of the system.
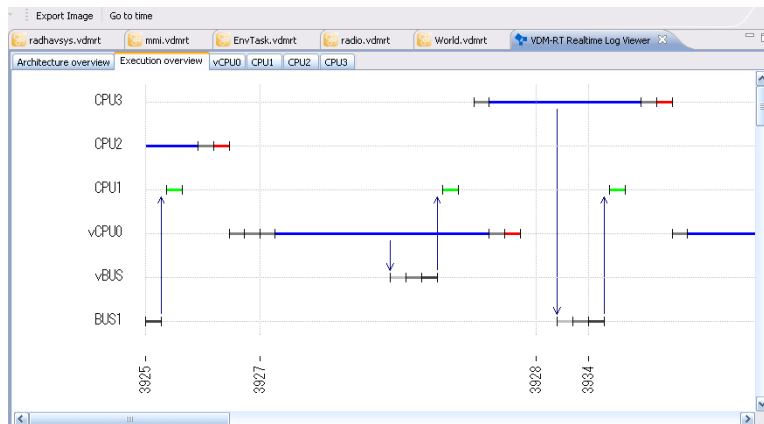


Figure 19: Execution overview

Since the complete execution of the model cannot be shown in a normal sized window, the user has the option of jumping to a certain time using the *Go to time* button. It is also possible to export all the generated views to *JPG* format using the *Export Image* button. All the generated pictures will be placed in the "logs" folder.

In addition to the execution overview, the RealTime Log Viewer can also give an overview of all executions on a single CPU. This view gives a detailed description of all operations and functions invoked on the CPU as well as the scheduling of concurrent processes. This can be seen in Figure 20.
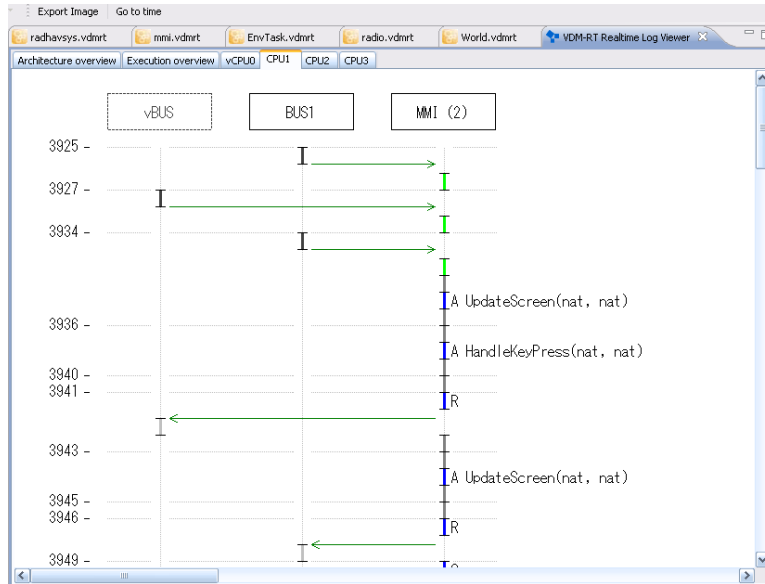
Figure 20: Execution on single CPU

# 14 Defining Your Own Java Libraries to be used from Overture

VDM models are not appropriate for describing everything and in addition it is common to have existing legacy code that a user may not wish to spend time on modelling use VDM but would like to make use of from a VDM model. Therefore a feature enabling a combination of a VDM model and a library with functionality in a standard `jar` file has been enabled. Using this feature it is possible from a VDM model to call functionality provided by the jar file. This functionality corresponds to DL modules/classes in a VDMTools context [**?**].

The external libraries which can be defined for VDM is coupled to the `is not yet specified` statements/ expressions. Only operations or functions of modules or classes can be delegated to an external jar. The look up for a delegate java class is done based on the module/class name where `_` is replaced to dot: `remote_lib_sensor` becomes `remote.lib.sensor` in java. The look up is only done once and only when a `is not yet specified` statements/ expressions is reached. The jar with the compiled remote library must be places in the project in a folder named `lib` it will then be put in the class-path of the interpreter when debugging. To compile a library jar for VDM the VDMJ jar needs to be in the class-path. It can be fund in the Overture IDE installation under `plugins/org.overture.ide.generated.vdmj_2.0.1/lib`.

## 14.1 Remote library example

In this example a remote sensor will be defined which could read a value from a real sensor. The VDM model interface of the sensor can be seen in listing 1 where as the Java class implementing

it can be seen in listing 2

```
class remote_lib_sensor

operations

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

end remote_lib_sensor
```

Listing 1: Remote sensor VDM class

```
package remote.lib;

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

public class sensor
{
        public Value getValue(Value id) throws ValueException
        {
                int intId = new Long(id.intValue(null)).intValue();
                return new IntegerValue(intId);
        }
}
```

Listing 2: Remote sensor Java class

# 15  Enabling Remote Control of the Overture Interpreter

In some situations it may also be valuable to be able to establish a front end (for example with a GUI) to interpreting different aspects of a VDM model. This is enabled in a fashion similar to in the previous section using a java jar file. This functionality corresponds to the CORBA based API from VDMTools [?].

A VDM model can be remote controlled by the use of the java interface RemoteControl. Remote control should be understood as a delegation of control of an interpretation, this means that the remote controller controls the debug session, it is responsible for taking action and executing parts of the VDM model when finished it should return and the debug session will stop. When a Remote controller is used the debugger continues working normally where breakpoints can be used. Launching a debugging session with the use of a remote controller can be done by placing the jar with the remote controller in a folder lib inside the project. And then specifying the full qualified name of the remote controller in the launch configuration under the *Remote Control* group. To compile a remote control jar for VDM the VDMJ jar needs to be in the class-path. It can be fund in the Overture IDE installation under

`plugins/org.overture.ide.generated.vdmj_2.0.1/lib.`

## 15.1   Example of a Remote Control class

In this example we have a VDM class A which just returns the input. As seen in listing 3 it is posible too call `execute` on the interpreter which is parsed to the remote controller which then returns a string with the result. An more advanced `valueExecute` also exists which returns the internal Value type of the interpreter which is use full for more advanced types.

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

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

Listing 3: Remote Controller Java class

# 16   A Command-Line Interface to VDMJ

A central part of the Overture tool is a Java application called VDMJ that provides a command-line interface that may be valuable for users outside the Eclipse interface of Overture.

## 16.1   Starting VDMJ

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

```
$ java -jar vdmj-2.0.0.jar
VDMJ: You must specify either -vdmsl, -vdmpp or -vdmrt
Usage: VDMJ <-vdmsl | -vdmpp | -vdmrt> [<options>] [<files>]
```

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

**-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 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. This is to allow a specification written in languages other than the default for your system to be used (see section **??**).

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

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

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

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

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

**-dtc:** this will disable all dynamic type checking.

**-log:** This will enable VDM-RT real-time event logging. These are useful with the Overture Eclipse GUI, which has a plugin to display timing diagrams (see Section 13).

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

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

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

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

## 16.2   Parsing, Type Checking, and Proof Obligations

All specification files loaded by VDMJ are parsed and type checked automatically. There are no type checking options; the type checker always uses "possible" semantics. If a specification does not parse and type check cleanly, the interpreter cannot be started and proof obligations cannot be generated (though warnings are allowed).

All warnings and error messages are printed on standard output, even with the -q option. A source file may contain VDM embedded in a LaTeX file using vdm_al environments (see Section 8); the markup is ignored by the parser, though reported line numbers will be correct.

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

## 16.3   The Interpreter with Debugging Fuctionality

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

```
$ java -jar vdmj-2.0.0.jar -vdmsl -i shmem.vdmsl
Parsed 1 module in 0.266 secs. No syntax errors
Type checked in 0.047 secs. No type errors
Interpreter started
```

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

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

**classes:** This command lists the loaded class names in both 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 in VDM-RT models. It starts to log real-time events to the file indicated. By default, event logging is turned off, but logging can be enabled to the console by using log with no arguments, or to a file using textttlog ¡filename¿. Logging can subsequently be turned off again by using textttlog off. The events logged include requests, activations and completions of all functions and operations, as well as all creation of instances of classes, creation of CPUs and BUSses, deployment of objects to specific CPUs and the swapping in an out of threads.

**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.

**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. Note that if combinatorial traces are defined in the model using the syntax from Section 10.1 then the print command can also be used to evaluate a named trace. This will carry out the combinatorial test by expanding all the regular expressions and executing the resulting operation sequences.

**assert <file>:** This command runs assertions from the file provided. The assertions in the file must be Boolean expressions, one per line. This command evaluates each assertion in turn, 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 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:** This command generates a list of all proof obligations for the VDM model that is loaded.

**break [<file>:]<line#> [<condition>]:** This command create 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 a function or an operation and optionally makes it a conditional breakpoint.

**trace [<file>:]<line#> [<exp>]:** This command creates a tracepoint for a specific line inside one of the source files. A tracepoint is similar to a breakpoint but with an implcit continue after it. This creates a trace of the expression given whenever the tracepoint is reached.

**trace <function/operation> [<exp>]:** This command create a tracepoint at the start of a function or operation. 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 [<file>|clear]:** This command displays/clears file line test coverage. 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, respectively. Finally, the percentage coverage of each source file is displayed.

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

**reload:** This command will re-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 reload.

**load <files>:** This command replace 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 load.

**[q]uit:** This command leave the interpreter.

When the interpretation 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.

**[n]ext:** This command is similar to textttstep 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 terminate the execution immediately.

# A   Templates in Overture

# B    Internal Errors

This appendix provides a list of the internal errors used in Overture and an explanantion, for each of them, the circumstances under which the internal error can be expected. Most of these errors should never be seen by an ordinary user, so if they appear please report it to the SourceForge bug reporting utility (`https://sourceforge.net/tracker/?group_id=141350&atid=749152`).

**0000:** File IO errors, e.g. `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`

**0012:** `Document has no specifications?`

**0013:** `Document has no expression?`

**0014:** `Unexpected type in definition block`

**0015:** `Unexpected type definition shape: <type>`

**0016:** `Typeless functions not supported` In an experimental version of Overture typeless functions was supported but this have been removed again.

**0017:** `Unexpected function shape: <shape>`

**0018:** `Unknown function body type`

**0019:** `Unexpected operation shape: <shape>`

**0020:** `Unknown operation body type`

**0021:** `Unknown instance variable type`

**0022:** `Unknown sync predicate type`

**0023:** `Expecting integer periodic argument`

**0024:** `Sporadic threads not implemented.` In the PhD thesis from Marcel Verhoef a notion of sporatic threads are included. However these are not (yet) incorporated into Overture.

**0025:** `Unknown thread specification type`

**0026:** `Let binding expects value definition`

**0027:** `Bare Dcl statement encountered`

**0028:** `Unknown trace specification type`

**0029:** `DBGP: <reason>`. This error is related to the protocol used between the GUI part of the debugger inside Eclipse and the underlying interpreter implementation inside VDMJ.

**0030:** `Statement type unsupported:  <type>`

**0031:** `Expected object state designator type`

**0032:** `Expected object state designator type`

**0033:** `Expected state designator type`

**0034:** `Native library error`

**0035:** `Expression type unsupported:  <type>`

**0036:** `Unexpected pattern/bind type`

**0037:** `Unexpected pattern/bind type`

**0038:** `Unexpected pattern/bind type`

**0039:** `Unexpected bind type`

**0040:** `Unexpected bind type`

**0041:** `Expected set bind type`

**0042:** `Expected set bind type`

**0043:** `Operator type unsupported:  <type>`

**0044:** Tuple field select is not a number

**0045:** Unexpected expression type:  <type>

**0046:** Unexpected literal expression

**0047:** Class instantiation not supported

**0048:** Unexpected type expression

**0049:** Unexpected literal pattern type

**0050:** Unexpected pattern type

**0051:** Unexpected scope value

**0052:** Cannot set default name at breakpoint

# 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`

# D  Syntatic Errors

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

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

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

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

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

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

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

**2006:** `Found tokens after class definitions`

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**2023:** Expecting '(' after operation name

**2024:** Expecting external declarations after 'ext'

**2025:** Expecting <name>:  exp->exp in errs clause

**2026:** Expecting 'rd' or 'wr' after 'ext'

**2027:** Expecting +ive number in periodic statement

**2028:** Expecting 'per' or 'mutex'

**2029:** Expecting <set bind> = <expression>

**2030:** Expecting simple field identifier

**2031:** Expecting field number after .#

**2032:** Expecting field name

**2033:** Expected 'is not specified' or 'is subclass responsibility'

**2034:** Unexpected token in expression

**2035:** Tuple must have >1 argument

**2036:** Expecting mk_<type>

**2037:** Malformed mk_<type> name <name>

**2038:** Expecting is_<type>

**2039:** Expecting maplet in map enumeration

**2040:** Expecting 'else' in 'if' expression

**2041:** Expecting two arguments for 'isofbase'

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

**2043:** Expecting two arguments for 'isofclass'

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

**2045:** Expecting two expressions in 'samebaseclass'

**2046:** Expecting two expressions in 'sameclass'

**2047:** Can't use history expression here

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

**2049:** Expecting 'end <module>'

**2050:** Expecting library name after 'uselib'

**2051:** Expecting 'end <module>'

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

**2053:** Exported function is not a function type

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

**2055:** Imported function is not a function type

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

**2057:** Unexpected token in pattern

**2058:** Expecting identifier

**2059:** Expecting a name

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

**2061:** Expecting a name

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

**2063:** Unexpected token in statement

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

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

**2066:** Expecting object field name

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

**2068:** Expecting field identifier

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

**2070:** Function type cannot return void type

**2071:** Expecting field identifier before ':'

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

**2073:** Duplicate field names in record type

**2074:** Unexpected token in type expression

**2075:** Expecting 'is subclass of'

**2076:** Expecting 'is subclass of'

**2077:** Expecting 'end' after class members

**2078:** Missing ';' after type definition

**2079:** Missing ';' after function definition

**2080:** Missing ';' after state definition

**2081:** Missing ';' after value definition

**2082:** Missing ';' after operation definition

**2083:** Expecting 'instance variables'

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

**2085:** Missing ';' after thread definition

**2086:** Missing ';' after sync definition

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

**2088:** Expecting '@' before type parameter

**2089:** Expecting '@' before type parameter

**2090:** Expecting ']' after type parameters

**2091:** Expecting ')' after function parameters

**2092:** Expecting '==' after parameters

**2093:** Missing colon after pattern/type parameter

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

**2095:** Implicit function must have post condition

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

**2097:** Expecting 'of' after state name

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

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

**2100:** Expecting 'end' after state definition

**2101:** Expecting ')' after operation parameters

**2102:** Expecting '==' after parameters

**2103:** Missing colon after pattern/type parameter

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

**2105:** Implicit operation must define a post condition

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

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

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

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

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

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

**2112:** Expecting '(' after periodic

**2113:** Expecting ')' after period arguments

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

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

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

**2117:** Expecting '(' after mutex

**2118:** Expecting ')' after 'all'

**2119:** Expecting ')'

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

**2121:** Expecting ')' after subsequence

**2122:** Expecting ')' after function args

**2123:** Expecting ']' after function instantiation

**2124:** Expecting ')'

**2125:** Expecting 'is not yet specified

**2126:** Expecting 'is not yet specified

**2127:** Expecting 'is subclass responsibility'

**2128:** Expecting comma separated record modifiers

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

**2130:** Expecting ')' after mu maplets

**2131:** Expecting ')' after mk_ tuple

**2132:** Expecting is_(expression, type)

**2133:** Expecting ')' after is_ expression

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

**2135:** Expecting '}' in empty map

**2136:** Expecting '}' after set comprehension

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

**2138:** Expecting '}' after set range

**2139:** Expecting '}' after set enumeration

**2140:** Expecting '}' after map comprehension

**2141:** Expecting '}' after map enumeration

**2142:** Expecting ']' after list comprehension

**2143:** Expecting ']' after list enumeration

**2144:** Missing 'then'

**2145:** Missing 'then' after 'elseif'

**2146:** Expecting ':'  after cases expression

**2147:** Expecting '->' after others

**2148:** Expecting 'end' after cases

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

**2150:** Expecting 'in' after local definitions

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

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

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

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

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

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

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

**2158:** Expecting 'in' after equals definitions

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

**2160:** Expecting '(' after 'isofbase'

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

**2162:** Expecting '(' after 'isofclass'

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

**2164:** Expecting '(' after 'samebaseclass'

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

**2166:** Expecting '(' after 'sameclass'

**2167:** Expecting ')' after 'sameclass' args

**2168:** Expecting <#op>(name(s))

**2169:** Expecting <#op>(name(s))

**2170:** Expecting 'module' at module start

**2171:** Expecting 'end' after module definitions

**2172:** Expecting 'dlmodule' at module start

**2173:** Expecting 'end' after dlmodule definitions

**2174:** Malformed imports?  Expecting 'exports' section

**2175:** Expecting ':'  after export name

**2176:** Expecting ':'  after export name

**2177:** Expecting ':'  after export name

**2178:** Expecting 'imports'

**2179:** Expecting 'from' in import definition

**2180:** Mismatched brackets in pattern

**2181:** Mismatched braces in pattern

**2182:** Mismatched square brackets in pattern

**2183:** Expecting '(' after mk_ tuple

**2184:** Expecting ')' after mk_ tuple

**2185:** Expecting '(' after <type> record

**2186:** Expecting ')' after <type> record

**2187:** Expecting 'is not yet specified

**2188:** Expecting 'is not yet specified

**2189:** Expecting 'is subclass responsibility'

**2190:** Expecting 'exit'

**2191:** Expecting 'tixe'

**2192:** Expecting '{' after 'tixe'

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

**2194:** Expecting 'in' after tixe traps

**2195:** Expecting 'trap'

**2196:** Expecting 'with' in trap statement

**2197:** Expecting 'in' in trap statement

**2198:** Expecting 'always'

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

**2200:** Expecting '||'

**2201:** Expecting '(' after '||'

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

**2203:** Expecting 'atomic'

**2204:** Expecting '(' after 'atomic'

**2205:** Expecting ')' after atomic assignments

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

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

**2208:** Expecting 'while'

**2209:** Expecting 'do' after while expression

**2210:** Expecting 'for'

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

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

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

**2214:** Expecting 'in' after pattern bind

**2215:** Expecting 'do' before loop statement

**2216:** Expecting '=' after for variable

**2217:** Expecting 'to' after from expression

**2218:** Expecting 'do' before loop statement

**2219:** Missing 'then'

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

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

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

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

**2224:** Expecting statement block

**2225:** Expecting ';' after statement

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

**2227:** Expecting ';' after declarations

**2228:** Expecting name:type in declaration

**2229:** Expecting 'return'

**2230:** Expecting 'let'

**2231:** Expecting 'in' after local definitions

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

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

**2234:** Expecting 'cases'

**2235:** Expecting ':'  after cases expression

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

**2237:** Expecting '->' after others

**2238:** Expecting 'end' after cases

**2239:** Expecting 'def'

**2240:** Expecting 'in' after equals definitions

**2241:** Expecting '['

**2242:** Expecting ']' after specification statement

**2243:** Expecting 'start'

**2244:** Expecting 'start('

**2245:** Expecting ')' after start object

**2246:** Expecting 'startlist'

**2247:** Expecting 'startlist('

**2248:** Expecting ')' after startlist objects

**2249:** Missing 'of' in compose type

**2250:** Missing 'end' in compose type

**2251:** Expecting 'to' in map type

**2252:** Expecting 'to' in inmap type

**2253:** Expecting 'of' after set

**2254:** Expecting 'of' after seq

**2255:** Expecting 'of' after seq1

**2256:** Bracket mismatch

**2257:** Missing close bracket after optional type

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

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

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

**2261:** Missing comma between return types?

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

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

**2264:** Expecting ':'  after trace name

**2265:** Expecting '{n1, n2}' after trace definition

**2266:** Expecting '{n}' or '{n1, n2}' after trace definition

**2267:** Expecting 'id.id(args)' or '(trace definitions)'

**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

**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

# E   Type Errors and Warnings

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

**3000:** `Expression does not match declared type`

**3001:** `Class inherits thread definition from multiple supertypes`

**3002:** `Circular class hierarchy detected:  <name>`

**3003:** `Undefined superclass:  <supername>`

**3004:** `Superclass name is not a class:  <supername>`

**3005:** `Overriding a superclass member of a different kind:  <member>`

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

**3007:** `Overriding member incompatible type:  <member>`

**3008:** `Overloaded members indistinguishable:  <member>`

**3009:** `Circular class hierarchy detected:  <class>`

**3010:** `Name <name> is ambiguous`

**3011:** `Name <name> is multiply defined in class`

**3012:** `Type <name> is multiply defined in class`

**3013:** `Class invariant is not a boolean expression`

**3014:** `Expression is not compatible with type bind`

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

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

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

**3018:** `Function returns unexpected type`

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

**3020:** `Too many parameter patterns`

**3021:** `Too few parameter patterns`

**3022:** `Too many curried parameters`

**3023:** `Too many parameter patterns`

**3024:** `Too few parameter patterns`

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

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

**3027:** `Operation returns unexpected type`

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

**3029:** `Function returns unexpected type`

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

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

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

**3035:** `Operation returns unexpected type`

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

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

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

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

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

**3041:** `Duplicate mutex name`

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**3059:** `Too many arguments`

**3060:** `Too few arguments`

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

**3062:** `Too many arguments`

**3063:** `Too few arguments`

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

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

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

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

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

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

**3070:** Right hand of function 'comp' is not a function

**3071:** Left hand function must have a single parameter

**3072:** Right hand function must have a single parameter

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

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

**3075:** Argument of 'conc' is not a seq of seq

**3076:** Argument of 'dinter' is not a set of sets

**3077:** Merge argument is not a set of maps

**3078:** dunion argument is not a set of sets

**3079:** Left of '<-:' is not a set

**3080:** Right of '<-:' is not a map

**3081:** Restriction of map should be set of <type>

**3082:** Left of '<:' is not a set

**3083:** Right of '<:' is not a map

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

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

**3086:** Else clause is not a boolean

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

**3088:** Predicate is not boolean

**3089:** Predicate is not boolean

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

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

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

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

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

**3095:** Field number does not match tuple size

**3096:** Argument to floor is not numeric

**3097:** Predicate is not boolean

**3098:** Function value is not polymorphic

**3099:** Polymorphic function is not in scope

**3100:** Function has no type parameters

**3101:** Expecting <n> type parameters

**3102:** Parameter name <name> not defined

**3103:** Function instantiation does not yield a function

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

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

**3106:** <operation> is not in scope

**3107:** Cannot use history of a constructor

**3108:** If expression is not a boolean

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

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

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

**3112:** Iota set bind is not a set

**3113:** Unknown type name <name>

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

**3115:** Undefined class type:  <class>

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

**3117:** Such that clause is not boolean

**3118:** Predicate is not boolean

**3119:** Map composition is not a maplet

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

**3121:** Element is not of maplet type

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

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

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

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

**3126:** Unknown type <type> in constructor

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

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

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

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

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

**3132:** mu operation on non-record type

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

**3134:** Class has no constructor with these parameter types

**3135:** Class has no constructor with these parameter types

**3136:** Left and right of '<>' different types

**3137:** Not expression is not a boolean

**3138:** Argument of 'not in set' is not a set

**3139:** Left hand of <operator> is not numeric

**3140:** Right hand of <operator> is not numeric

**3141:** Right hand of '++' is not a map

**3142:** Right hand of '++' is not a map

**3143:** Domain of right hand of '++' must be nat1

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

**3145:** Argument to 'power' is not a set

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

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

**3148:** Left of ':->' is not a map

**3149:** Right of ':->' is not a set

**3150:** Restriction of map should be set of <type>

**3151:** Left of ':>' is not a map

**3152:** Right of ':>' is not a set

**3153:** Restriction of map should be set of <type>

**3154:** <name> not in scope

**3155:** List comprehension must define one numeric bind variable

**3156:** Predicate is not boolean

**3157:** Left hand of 'ˆ' is not a sequence

**3158:** Right hand of 'ˆ' is not a sequence

**3159:** Predicate is not boolean

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

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

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

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

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

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

**3166:** Set range type must be an number

**3167:** Set range type must be an number

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

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

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

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

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

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

**3174:** Subsequence is not of a sequence type

**3175:** Subsequence range start is not a number

**3176:** Subsequence range end is not a number

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

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

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

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

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

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

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

**3184:** Exported <name> function type incorrect

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

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

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

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

**3189:** Exported type does not match actual type

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

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

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

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

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

**3195:** Cannot import from self

**3196:** No such module as <module>

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

**3198:** Type bind not compatible with expression

**3199:** Set bind not compatible with expression

**3200:** Mk_ expression is not a record type

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

**3202:** Record pattern argument/field count mismatch

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

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

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

**3206:** Matching expression is not a set type

**3207:** Object designator is not an object type

**3208:** Object designator is not an object type

**3209:** Member <field> is not in scope

**3210:** Object member is neither a function nor an operation

**3211:** Expecting <n> arguments

**3212:** Unexpected type for argument <n>

**3213:** Operation <name> is not in scope

**3214:** Cannot call <name> from static context

**3215:** <name> is not an operation

**3216:** Expecting <n> arguments

**3217:** Unexpected type for argument <n>

**3218:** Expression is not boolean

**3219:** For all statement does not contain a set type

**3220:** From type is not numeric

**3221:** To type is not numeric

**3222:** By type is not numeric

**3223:** Expecting sequence type after 'in'

**3224:** If expression is not boolean

**3225:** Such that clause is not boolean

**3226:** Incompatible types in object assignment

**3228:** <name> is not in scope

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

**3230:** <name> is implicit

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

**3232:** <name> is not an operation name

**3233:** Precondition is not a boolean expression

**3234:** Postcondition is not a boolean expression

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

**3236:** Class does not define a thread

**3237:** Class does not define a thread

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

**3239:** Incompatible types in assignment

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

**3242:** Map element assignment of wrong type

**3243:** Seq element assignment is not numeric

**3244:** Expecting a map or a sequence

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

**3246:** Unknown field name, <name>

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

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

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

**3250:** Map application must have one argument

**3251:** Map application argument is incompatible type

**3252:** Sequence application must have one argument

**3253:** Sequence argument is not numeric

**3254:** Too many arguments

**3255:** Too few arguments

**3256:** Inappropriate type for argument <n>

**3257:** Too many arguments

**3258:** Too few arguments

**3259:** Inappropriate type for argument <n>

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

**3261:** Unknown field name, <name>

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

**3263:** Cannot reference 'self' from here

**3264:** At least one bind cannot match set

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

**3266:** Argument is not an object

**3267:** Empty map cannot be applied

**3268:** Empty sequence cannot be applied

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

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

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

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

**3273:** Measure not allowed for an implicit function

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

**3275:** Error clause must be a boolean

**3276:** Ambiguous names inherited by <name>

**3277:** Trace repeat illegal values

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

**3279:** Cannot instantiate system class <name>

**3280:** Argument to deploy must be an object

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

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

**3283:** System class constructor cannot be implicit

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

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

**3286:** Constructor cannot be 'async'

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

**3288:** Period argument must be non-zero

**3289:** Delay argument must be less than the period

**3290:** Argument to setPriority must be an operation

**3291:** Argument to setPriority cannot be a constructor

**3292:** Constructor is not accessible

**3293:** Asynchronous operation <name> cannot return a value

**3294:** Only one system class permitted

**3295:** Argument to 'reverse' is not a sequence

**3296:** `Cannot use <typename> outside system class`

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

**3298:** `Cannot inherit from CPU`

**3299:** `Cannot inherit from BUS`

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

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

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

Warnings from the type checker include:

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

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

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

**5003:** `Permission guard of overloaded operation`

**5004:** `History expression of overloaded operation`

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

**5006:** `Statement will not be reached`

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

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

**5009:** `Empty set used in bind`

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

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

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

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

# F Run-Time Errors

When using the interpreter/debugger it is possible to get run-time errors indicating that a problem with the VDM model analysed have been detected. This includes the following kinds of errors:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**4017:** `Duplicate map keys have different values:  <domain>`

**4018:** `Maplet cannot be evaluated`

**4019:** `Sequence cannot extend to key:  <index>`

**4020:** `State value is neither a <type> nor a <type>`

**4021:** `Duplicate map keys have different values:  <key>`

**4022:** `mk_ type argument is not <type>`

**4023:** `Mu type conflict?  No field tag <tag>`

**4024:** 'not yet specified' expression reached

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

**4026:** Cannot create post_op environment

**4027:** Cannot create pre_op environment

**4028:** Sequence comprehension pattern has multiple variables

**4029:** Sequence comprehension bindings must be numeric

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

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

**4032:** 'is subclass responsibility' expression reached

**4033:** Tail sequence is empty

**4034:** Name <name> not in scope

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

**4036:** ERROR statement reached

**4037:** No such field:  <name>

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

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

**4040:** Let be st found no applicable bindings

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

**4042:** Sequence does not contain key:  <key>

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

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

**4046:** No such field:  <name>

**4047:** Cannot execute specification statement

**4048:** 'is subclass responsibility' statement reached

**4049:** Value <value> is not in set bind

**4050:** Value <value> is not in set bind

**4051:** Cannot apply implicit function: <name>

**4052:** Wrong number of arguments passed to <name>

**4053:** Parameter patterns do not match arguments

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

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

**4057:** Curried function return type is not a function

**4058:** Value <value> is not a nat1

**4059:** Value <value> is not a nat

**4060:** Type invariant violated for <type>

**4061:** No such key value in map: <key>

**4062:** Cannot convert non-injective map to an inmap

**4063:** Duplicate map keys have different values: <domain>

**4064:** Value <value> is not a nat1 number

**4065:** Value <value> is not a nat

**4066:** Cannot call implicit operation: <name>

**4067:** Deadlock detected

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

**4069:** Parameter patterns do not match arguments

**4071:** Precondition failure: <pre_name>

**4072:** Postcondition failure: <post_name>

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

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

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

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

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

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

**4079:** Type invariant violated by mk_ arguments

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

**4081:** Field not defined:  <tag>

**4082:** Type invariant violated by mk_ arguments

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

**4084:** Cannot convert empty sequence to seq1

**4085:** Cannot convert tuple to <type>

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

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

**4088:** Set not permitted for <kind>

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**4106:** Boolean pattern match failed

**4107:** Character pattern match failed

**4108:** Sequence concatenation pattern does not match expression

**4109:** Values do not match concatenation pattern

**4110:** Expression pattern match failed

**4111:** Integer pattern match failed

**4112:** Quote pattern match failed

**4113:** Real pattern match failed

**4114:** Record type does not match pattern

**4115:** Record expression does not match pattern

**4116:** Values do not match record pattern

**4117:** Wrong number of elements for sequence pattern

**4118:** Values do not match sequence pattern

**4119:** Wrong number of elements for set pattern

**4120:** Values do not match set pattern

**4121:** Cannot match set pattern

**4122:** String pattern match failed

**4123:** Tuple expression does not match pattern

**4124:** Values do not match tuple pattern

**4125:** Set union pattern does not match expression

**4126:** Values do not match union pattern

**4127:** Cannot match set pattern

**4129:** Exit <value>

**4130:** Instance invariant violated:  <inv_op>

**4131:** State invariant violated:  <inv_op>

**4132:** Using undefined value

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

**4134:** Infinite or NaN trouble

**4135:** Cannot instantiate a system class

**4136:** Cannot deploy to CPU

**4137:** Cannot set operation priority on CPU

**4138:** Cannot set CPU priority for operation

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

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

**4141:** CPU policy does not allow priorities

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

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

**4144:** State init expression cannot be executed

**4145:** Time:  <n> is not a nat1

# G   Categores of Proof Obligations

This appendix provide a list of the different proof obligation categories used in Overture and an explanantion for each of them the circumstances under which the PO category can be expected.

**map apply:**  Whenever a map application is used it needs to be ensured that the argument is indeed in the domain of the mapping.

**function apply:**  Whenever a function application is used it needs to be ensured that the list of arguments to the function are all of the types expected by the function signature as well as satisfy the pre-condition of the function in case such a predicate is present.

**sequence apply:**  Whenever a sequence application is used it needs to be ensured that the argument is indeed in the indices of the sequence.

**post condition:**

**function satisfiability:**  For all implicit function definitions this proof obligation will be generated to ensure that it will be possible to find a result satisfying the post-condition for all arguments of the function input types satisfying the pre-conditions.

**function parameter patterns:**

**let be st existence:**  Whenever a let-be-such-that expression/statement is used it needs to be guranteed that the set to selecte from is non-empty.

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

**function iteration:**

**map iteration:**

**function compose:**

**map compose:**

**non-empty set:**  This kind of proof obligations are used whenever non-empty sets are required.

**non-empty sequence:**  This kind of proof obligations are used whenever non-empty sequences are required.

**non-zero:**  This kind of proof obligations are used whenever zero cannot be used (e.g. in division).

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

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

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

**map sequence compatible:**

**map set compatible:**

**sequence modification:**

**tuple selection:** This proof obligation category is used whenever a tuple selection expression is used and it must be guranteed that the length of the tuple at least is as long as the selector used.

**value binding:**

**subtype:** This proof obligation category is used whenever it is not possible to statically detect that the given expression indeed falls into the subtype required in the actual use of it.

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

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

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

**state invariant:** If a state (including instance variables in VDM++) have an invariant this proof obligation will be generated whenever assignment is made to a part of the state all the places where the invariant shall be satisfied.

**while loop termination:** This kind of proof obligation is a reminder to ensure that a while loop is terminating. However, for embedded systems that is typically not desirable and thus in those cases there is no need to satisfy this proof obligation.

**operation post condition:** Whenever an explicit operation has a post-condition there is an implicit proof obligation generated to remind the user that one would have to ensure that the explicit body of the operation for all possible input satisfy the post-condition.

**operation parameter patterns:**

**operation satifiability:** For all implicit operation definitions this proof obligation will be generated to ensure that it will be possible to find a result satisfying the post-condition for all arguments of the operation input types satisfying the pre-conditions.

# H   Adding external plug-ins to Overture IDE

The Overture IDE is build as a RCP[4] and there is no update manager included, however external plug-ins can still be manually installed. To install a plug-in there are two things which needs to be done:

1. Copy the plug-in jar or jars into the plugin folder of the Overture IDE installation.

2. Add the plug-in jar file names to the `osgi.bundles` in the config.ini file which is found under the folder configuration.

After the steps above has been performed the IDE can be started and the new plug-ins will be available. (Remember to include all depended plug-ins which is not already a part of the Overture IDE).

## H.1   Adding SVN support to the Overture IDE

Adding SVN support to the Overture IDE is rather simple:

1. Fetch the update site jars from `http://subclipse.tigris.org/` under Download and Install and select Zipped downloads. The latest version demonstrated in this guide is site-1.6.10.zip.

2. Copy the features and plugins folders into the Overture IDE folder, merging them with the ones of Overture.

3. Then add the plugins to the `osgi.bundles` in the confi.ini file separated by comma.

   - org.tigris.subversion.clientadapter
   - org.tigris.subversion.clientadapter_1.6.10.jar
   - org.tigris.subversion.clientadapter.javahl
   - org.tigris.subversion.clientadapter.javahl_1.6.9.3.jar
   - org.tigris.subversion.clientadapter.javahl.win32
   - org.tigris.subversion.clientadapter.javahl.win32_1.6.9
   - org.tigris.subversion.subclipse.core
   - org.tigris.subversion.subclipse.core_1.6.10.jar
   - org.tigris.subversion.subclipse.doc
   - org.tigris.subversion.subclipse.doc_1.3.0.jar
   - org.tigris.subversion.subclipse.graph
   - org.tigris.subversion.subclipse.graph_1.0.7.jar

---

[4]Rich Client Platform

- org.tigris.subversion.subclipse.ui
- org.tigris.subversion.subclipse.ui_1.6.10.jar

When the steps above is completed the Overture IDE can be started and SVN support will then be available.

# I Index