

Overture – Open-source Tools for Formal Modelling TR-2010-01
January 2010

Tutorial to Overture/VDM-SL

by

Peter Gorm Larsen
John Fitzgerald
Sune Wolff
Nick Battle
Kenneth Lausdahl
Augusto Ribeiro
Kenneth Pierce
Steve Riddle





Contents

3	An Introduction to Overture Tool Support for VDM-SL	1
3.1	Introduction	1
3.2	Getting Hold of the Software	2
3.3	Using the Overture Perspective	3
3.4	Getting Started using Templates	5
3.5	Debugging	8
3.5.1	The Debug configuration	9
3.6	Test coverage	12
3.7	Proof Obligations	13
3.8	A Command-Line Interface	14
3.9	Summary	16
A	A Chemical Plant Example	21
A.1	An informal description	21
A.2	A VDM-SL model of the Alarm example	22

Chapter 3

An Introduction to Overture Tool Support for VDM-SL

Preamble

This is an introduction to the Overture Integrated Development Environment (IDE) and its facilities for supporting modelling and analysis in VDM-SL. It may be used as a substitute for Chapter 3 of “Modelling Systems – Practical Tools and Techniques in Software Development”¹ or as a free-standing guide. Additional material is available on the book’s web site www.vdmbook.com. Throughout this guide we will refer to the textbook as “the book” and the book’s web site simply as “the web site”.

We will use examples based on the *alarm* case study introduced in Chapter 2 of the book. For readers using this as a free-standing guide, an informal explanation of the case study and its VDM-SL model are given in A.

We will introduce the Overture tools during a “hands-on” tour of their functionality, providing enough detail to allow you to use Overture for serious applications, including the exercises in the book. However, this is by no means a complete guide to Overture; more information can be obtained from

www.overturetool.org.

3.1 Introduction

Models in VDM are formal in the sense that they have a very precisely described semantics, making it possible to analyse models in order to confirm or refute claims about them. Such an analysis often reveals gaps in the developer’s and the client’s understanding of the system, allowing these to be resolved before an expensive commitment is made to program code. The process of analysing

¹John Fitzgerald and Peter Gorm Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*, Cambridge University Press, 2nd edition 2009.



claims about systems modelled in this way is termed *validation* and is discussed in greater depth in Chapter 10 of the book.

Section 3.2 describes how to obtain the Overture tools. Section 3.3 introduces terminology used by Eclipse-based tools like Overture. Section 3.4 describes the features that support the construction and editing of VDM-SL models. Section 3.5 describes the testing and debugging features and Section 3.6 describes how line coverage from using the debugger can be covered and displayed. Afterwards Section 3.7 describes the facilities for automatically generating the checks (called *proof obligations*) that must be performed in order to ensure that a VDM-SL model is consistent and meaningful. Finally, Section 3.8 shows how parts of Overture’s functionality can be accessed through a command line interface, allowing batch-mode testing.

3.2 Getting Hold of the Software

Overture is an open source tool, developed by a community of volunteers and built on 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 Mac². 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 it is selecting and tick off the box for “use this as the default and do not ask again”. A welcome screen will also only the first time introduce you to the overall mission of the Overture open source initiative.

A large library of sample VDM-SL models, including all those needed for the exercises in the book, is available and can be downloaded from SourceForge as the `examplesSL.zip` file using the URL³:

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

You can import the example library zip folder as described in Section 3.3. Finally, in order to make use of the test coverage feature described in Section 3.6 it is necessary to have \LaTeX and its `pdflatex` feature. This can for example be obtained from:

`http://miktex.org/2.8/`

²It 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.

³The library files are created to be used with Eclipse, but can be opened with file compression programs like Winrar on Windows.



Note for users of VDMTools®

Overture provides a new open source VDM tool set, but it can also work with VDMTools®. VDMTools, originally developed IFAD A/S, is now maintained and developed by CSK Systems (see <http://www.vdmttools.jp/en/>). In the future Overture will be able to access VDMTools functionality via a remote API, but the integration will to some extent be limited by the API capabilities. However, the additional features of the Overture IDE make it worth considering as a front end to the VDMTools functionality.

3.3 Using the Overture Perspective

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 3.1. A collection of panels is called a *perspective*, for example Figure 3.1 shows the standard Overture perspective. This consists of views for managing Overture projects and viewing and editing files in an Overture project. The perspectives available in Overture will be described later, but for the moment think of a perspective as a particular composition of views that is useful for conducting a particular task. Note that the first time Overture would like to automatically change to a specific perspective it will ask you for permission to do so.

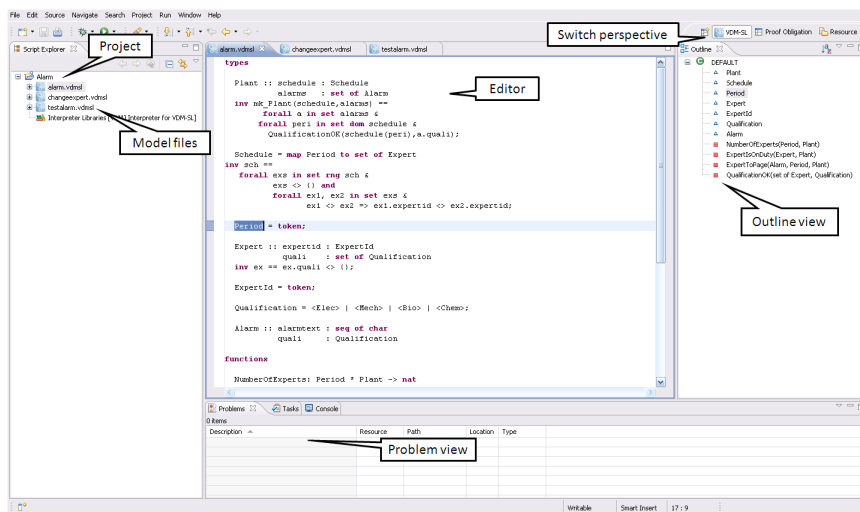


Figure 3.1: The Overture Perspective

The *Script Explorer* view allows you to create, select, and delete Overture projects and navigate between the files in these projects. Start by importing the alarm project. This can be done by right-clicking the explorer view and selecting *Import*, followed by *General* → *Existing Projects*



into *Workspace*. In this way the projects from `examplesSL.zip` file mentioned above can be imported very easily. Initially it is recommended that you only import the `AlarmErrSL` and the `AlarmSL` projects as shown in Figure 3.2⁴.

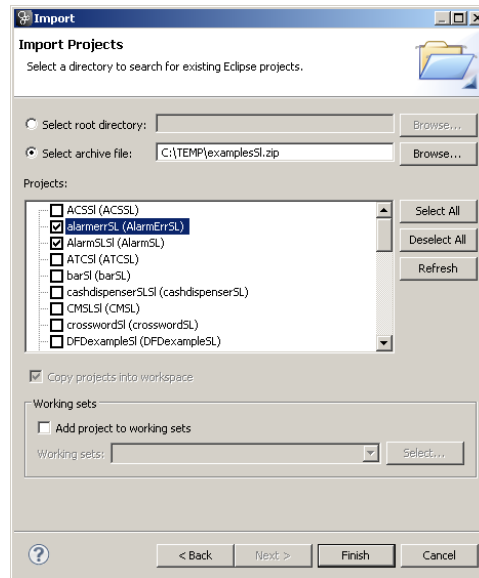


Figure 3.2: Importing the `Alarm` VDM-SL Projects

Depending on the dialect of VDM used in a given project, a corresponding Overture editor will be available here.

The *Outline view*, to the right of the editor (see Figure 3.3), presents an outline of the model in the file selected in the editor. It displays any declared VDM-SL modules, as well as their state components, values, types, functions and operations. In case of a flat VDM-SL model the module is called `DEFAULT`. Figure 3.1 shows the outline view on the right hand side. Clicking on an operation or function name will move the cursor in the editor to the definition of the operation or function. At the top of the outline view there is a button to determine what is displayed in the outline view (it is possible to hide different kinds of definitions, for example).

The *Problems view* displays messages about the projects on which you are working. This includes information generated by Overture, such as warnings and errors. Note that all errors and warnings also appear as tooltips in the VDM-SL editor.

Most of the other features of the workbench, such as the menus and toolbars, are similar to those of other Eclipse applications, apart from 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; the shortcuts vary dynamically depending on context and history.

⁴You need both of these for carrying out different kinds of exercises throughout this chapter.

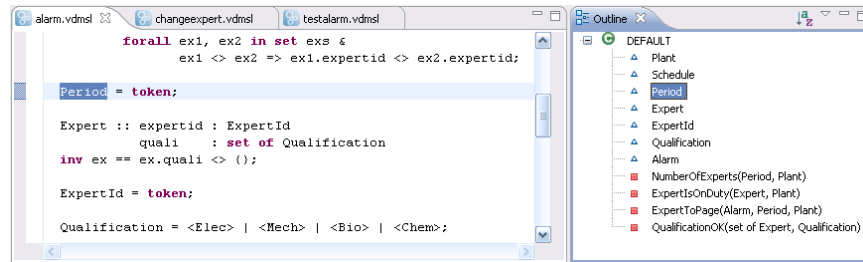


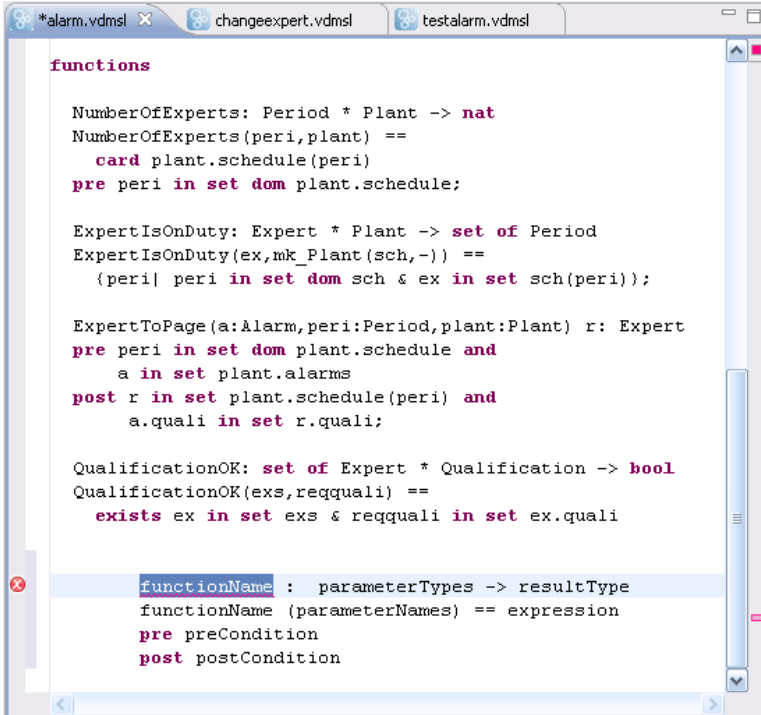
Figure 3.3: The Outline View

3.4 Getting Started using Templates

Before proceeding, please make sure that you have imported both the `AlarmErrSL` and the `AlarmSL` projects as shown in Figure 3.2. When editing a VDM-SL model, the Overture IDE parses the content of the editor buffer continuously as changes are made. Any parse errors will be reported in the problems view, as well as being highlighted in the editor. See the bottom of Figure 3.1. Each time a VDM-SL model file is saved the editor type-checks the model and reports any errors or warnings. Note also that the suggestions made in the error messages may not always be entirely the action you may wish to take when correcting the source since the tool cannot guess what you intended to write.

Templates can be particularly useful when modifying VDM-SL models. 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 3.4. 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.

A new VDM-SL project is created by choosing `File → New → Project`. The dialog box shown in Figure 3.5 will appear. Ensure that VDM-SL is selected as the project type, click *Next* and then name the project `Test` and if next is selected again it gets possible to select standard libraries as shown in Figure 3.6. These standard libraries require users to make use of modules but in return it is possible to get standard input/output, math and general utility functionality by selecting the appropriate standard libraries. In this `Test` project we can try to select the `IO` standard library. Afterwards one simply select *Finish*. Now you have an almost empty project (with the exception of the `IO.vdmsl` file in the `lib` directory) and you can then either add new VDM-SL files to the project or simply paste in existing VDM-SL source files from elsewhere. Adding a VDM-SL file to a project you can rightclick on the project and then select `New → VDM-SL Module` and then give a meaningful name (e.g. `Test`) to the module you would like to start defining and press *Finish*. This will create a new file with a module with the selected name and with space for the different kinds of definitions you can make inside such a VDM-SL module.



```
functions

NumberOfExperts: Period * Plant -> nat
NumberOfExperts(peri,plant) ==
  card plant.schedule(peri)
pre peri in set dom plant.schedule;

ExpertIsOnDuty: Expert * Plant -> set of Period
ExpertIsOnDuty(ex,mk_Plant(sch,-)) ==
  {peri| peri in set dom sch & ex in set sch(peri)};

ExpertToPage(a:Alarm,peri:Period,plant:Plant) r: Expert
pre peri in set dom plant.schedule and
  a in set plant.alarms
post r in set plant.schedule(peri) and
  a.quali in set r.quali;

QualificationOK: set of Expert * Qualification -> bool
QualificationOK(exs,reqquali) ==
  exists ex in set exs & reqquali in set ex.quali

functionName : parameterTypes -> resultType
functionName (parameterNames) == expression
pre precondition
post postCondition
```

Figure 3.4: Explicit function template

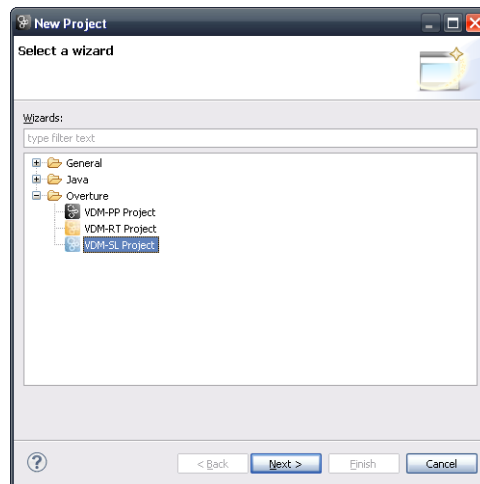


Figure 3.5: Creating a New VDM-SL Project

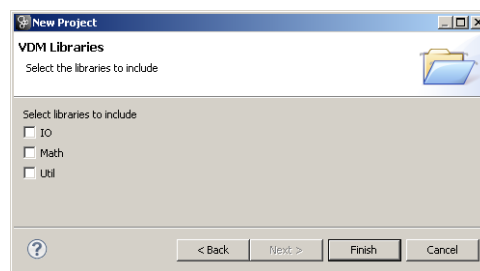


Figure 3.6: The VDM-SL Standard Libraries

Exercise 3.1 Try to create a new `Test` VDM-SL project and update the `test.vdmsl` file before **exports all** with:

```
imports from IO all
```

in order to make use of the `IO` standard library in the `Test` module. Inside `IO` there is for example a definition of a function called `print` and that can for example be used in an operation as:

```
Try: nat ==> nat
Try(n) ==
  (IO`print(n);
   return n + 1)
```

Insert this and later on when you have learned how to create a debug configuration you can try to



see what happens when `Try` is debugged. □

3.5 Debugging

This section describes facilities for debugging a model by stepping through the evaluation of functions and operations. The alarm example is used. The following test file (`testalarm.vdmsl`) can be found in the alarm project.

values

```
p1:Period = mk_token("Monday day");
p2:Period = mk_token("Monday night");
p3:Period = mk_token("Tuesday day");
p4:Period = mk_token("Tuesday night");
p5:Period = mk_token("Wednesday day");

eid1:ExpertId = mk_token(134);
eid2:ExpertId = mk_token(145);
eid3:ExpertId = mk_token(154);
eid4:ExpertId = mk_token(165);
eid5:ExpertId = mk_token(169);
eid6:ExpertId = mk_token(174);
eid7:ExpertId = mk_token(181);
eid8:ExpertId = mk_token(190);

e1:Expert = mk_Expert(eid1, {<Elec>});
e2:Expert = mk_Expert(eid2, {<Mech>, <Chem>});
e3:Expert = mk_Expert(eid3, {<Bio>, <Chem>, <Elec>});
e4:Expert = mk_Expert(eid4, {<Bio>});
e5:Expert = mk_Expert(eid5, {<Chem>, <Bio>});
e6:Expert = mk_Expert(eid6, {<Elec>, <Chem>, <Bio>, <Mech>});
e7:Expert = mk_Expert(eid7, {<Elec>, <Mech>});
e8:Expert = mk_Expert(eid8, {<Mech>, <Bio>});

s: map Period to set of Expert
  = {p1 |-> {e7, e5, e1},
     p2 |-> {e6},
     p3 |-> {e1, e3, e8},
     p4 |-> {e6}};
```



```

a1:Alarm = mk_Alarm("Power supply missing", <Elec>);
a2:Alarm = mk_Alarm("Tank overflow", <Mech>);
a3:Alarm = mk_Alarm("CO2 detected", <Chem>);
a4:Alarm = mk_Alarm("Biological attack", <Bio>);

plant1: Plant = mk_Plant(s, {a1, a2, a3, a4})

```

By using this test, it is possible to exercise the system in order to check whether, for this test, the correct expert is paged as a result of a given alarm.

3.5.1 The Debug configuration

Before the debugging can be initiated in Overture, a debug configuration must be created by right-clicking the project and choosing *Debug As* → *Debug configuration*⁵. The debug configuration dialog requires the project name, the class and the operation/function used as the entry point of the test. Figure 3.7 shows the debug configuration for the alarm model. The class and operation/function name can be chosen from a Browse dialog; if the operation or function has arguments, these must be typed in manually between the brackets of the entry point function/operation. If this is not types such that the overall expression is type correct an error will be shown at the top of the debug configuration window. This means that one need to change the *Operation* line for example from:

```
NumberOfExperts((Period) peri, (Plant) plant)
```

to for example:

```
NumberOfExperts(p2, plant1)
```

Once the configuration is ready, the model can be debugged. If one have already set a breakpoint on one of the lines that will be executed, this will change the main perspective of the Overture IDE to the *Debug perspective*. If no breakpoints are set the result is simply shown in the *Debug Console* view in the lower part of the Overture perspective. Breakpoints can easily be set by double clicking in left margin in the editor view. When the debugger reaches the location of a breakpoint, evaluation suspends and the user can inspect the values of variables and step through the VDM-SL model line by line. So for `NumberOfExperts` try to set such a breakpoint in line 34 and debug again.

The Debug perspective is illustrated in Figure 3.8. The *Debug view* is located in the upper left corner of the Debug perspective. It shows all running models and the call stacks belonging to them. It also shows whether a given model is stopped, suspended or running.

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 3.1).

⁵Note that the *Run As* functionality existing Eclipse users are used to is not supported in the current version of Overture.

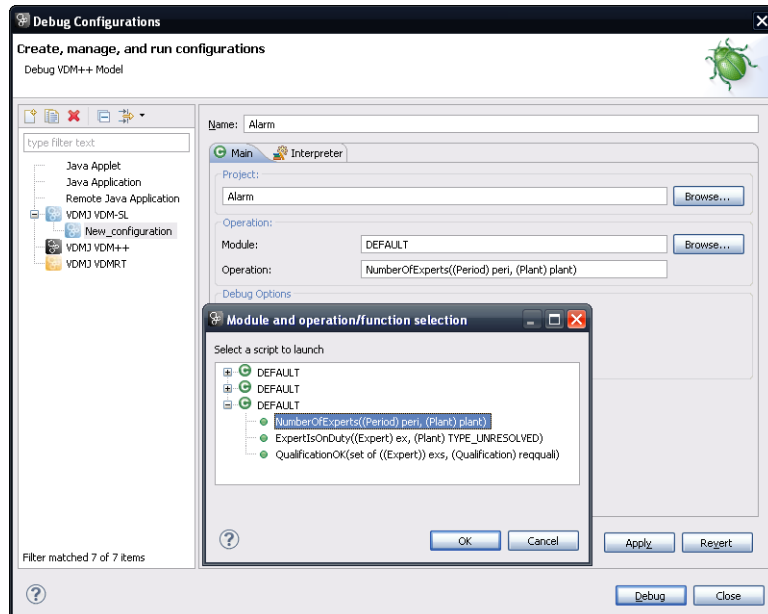


Figure 3.7: The debug configuration dialog

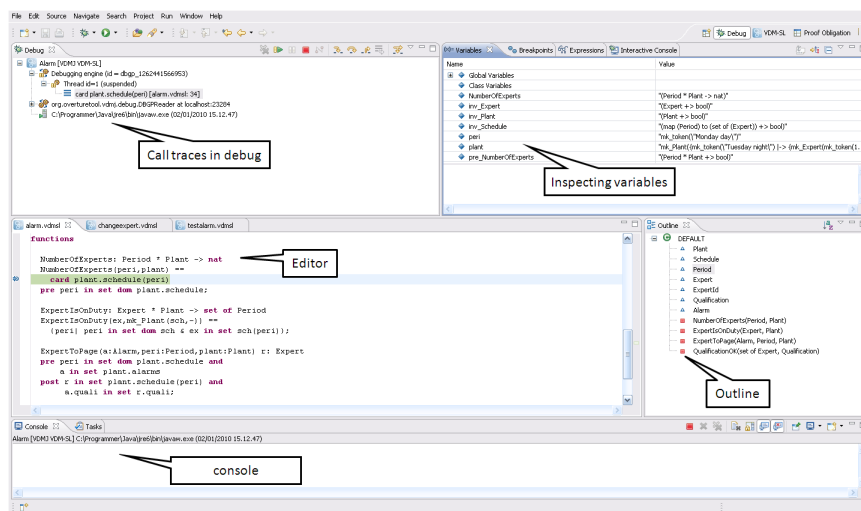


Figure 3.8: Debugging perspective

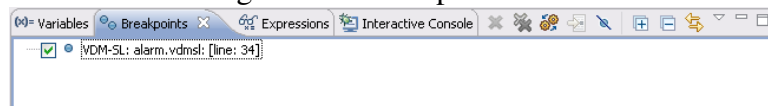


Table 3.1: Overture debugging buttons

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

The *Variables view* shows all the variables in a given context when a breakpoint is reached. The variables and their displayed values are automatically updated when stepping through a model. The view is located in the upper right hand corner in the Debug perspective.

Figure 3.9: Breakpoint View



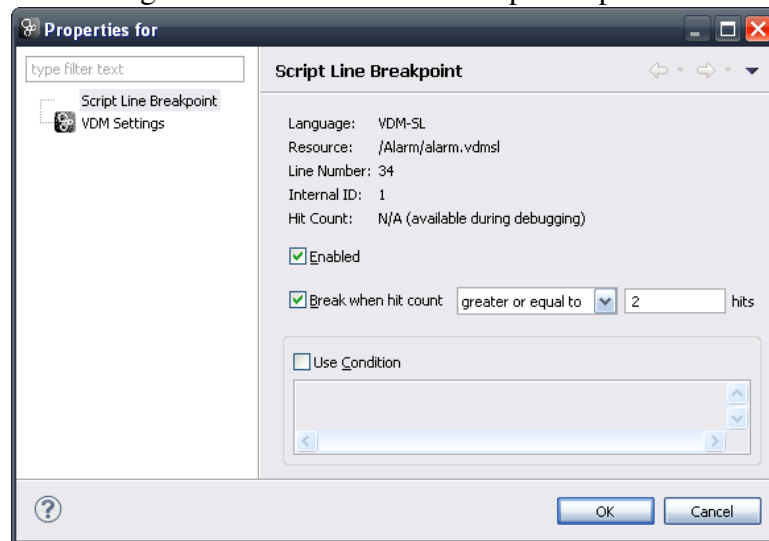
The *Breakpoints view* gives an overview of all breakpoints set (see Figure 3.9). From this view you can navigate to the location of a given breakpoint, disable or delete it, or set its properties. Conditional breakpoints are supported. These are a powerful tool for the developer since they allow you to specify a condition that must be true in order for the debugger to stop at the given breakpoint. The condition can either be a boolean expression using variables in scope at the breakpoint, or it can be a hit count after which the breakpoint should become active.

You can make a simple breakpoint conditional by right clicking on the breakpoint mark in the left margin of the editor and selecting the option *Breakpoint properties*. This opens a dialog shown in Figure 3.10.

The *Expressions view* allows you to enter *watch* expressions whose values are automatically displayed and updated when stepping. “Watch expressions” can be added manually or created by



Figure 3.10: Conditional breakpoint options



selecting *create watch expression* from the Variables view. It is possible to edit existing expressions. Like the Breakpoints view, this view is hidden in the upper right hand corner in Figure 3.8.

While the Expressions view allows you to inspect values, its functionality is somewhat limited. For more thorough inspections, the *Interactive Console* view is provided. Here commands can be executed in a given context, i.e. when the debugger is at a breakpoint. The Interactive Console keeps a command history so that previously executed commands can easily be run again. The interactive console can be seen at the bottom of Figure 3.8.

Exercise 3.2 Use the interpreter to evaluate the following expressions:

```
NumberOfExperts(p2,plant1)
NumberOfExperts(p3,plant1)
ExpertIsOnDuty(e1,plant1)
ExpertIsOnDuty(e2,plant1)
ExpertIsOnDuty(e3,plant1)
```

□

3.6 Test coverage

It is often useful to know how much of a model has been exercised by a set of tests. This gives some insight into the thoroughness of a test suite and may also help to identify parts of the model that have not been assessed, allowing new tests to be devised to cover these. When any evaluation



is performed on a VDM-SL model, the interpreter records the lines of the VDM-SL model that are executed. This permits the line coverage to be examined after a test to identify the parts of the VDM-SL model that have not yet been exercised – coverage is cumulative, so a set of tests can be executed and their total coverage examined at the end.

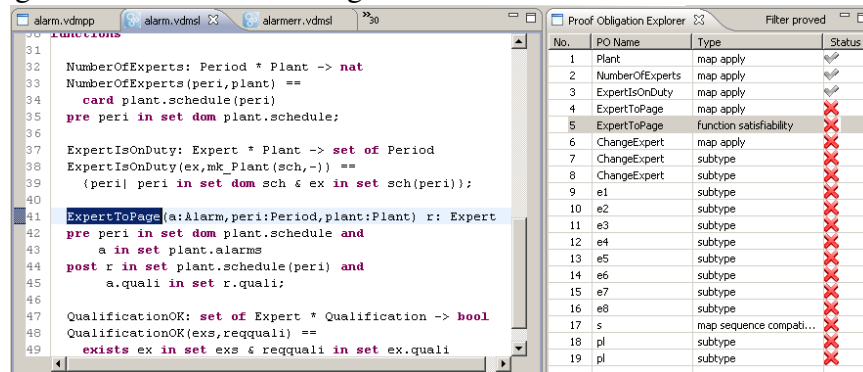
In our simple example, the different tests in the exercise above does cause the majority of the VDM-SL model to be executed, but the `ExpertIsOnDuty` function in `alarm.vdmsl` is covered.

3.7 Proof Obligations

Overture can also generate the *Proof Obligations* for a VDM-SL model. These are boolean expressions which highlight areas of the VDM-SL model where some constraint must be met in order to guarantee internal consistency (i.e. no run-time errors will occur while debugging if these are all satisfied). This includes type and class invariants or function or operation pre/post conditions. Each proof obligation should evaluate to *true*. Understanding these in detail are explained further in Chapter 10.

This feature is invoked by right clicking on the project in the *Explorer view* and then selecting the *Proof Obligations* → *Generate Proof Obligations* entry. This will start up a proof obligation perspective with a special *PO view*. For the alarm example this view looks like in Figure 3.11. Here

Figure 3.11: The Proof Obligation view for the Alarm VDM-SL model



the fifth proof obligation is related to the satisfiability of the `ExpertToPage` function which is defined as follows:

```
ExpertToPage(a:Alarm,peri:Period,plant:Plant) r: Expert
  pre peri in set dom plant.schedule and
    a in set plant.alarms
  post r in set plant.schedule(peri) and
    a.quali in set r.quali;
```




Here the proof obligation records the constraint that, for all possible arguments satisfying the pre-condition, the post-condition allows at least one possible valid result of the function. This is described as a proof obligation as follows:

```
(forall a:Alarm, peri:Period, plant:Plant &
  pre_ExpertToPage(a, peri, plant) =>
  exists r:Expert & post_ExpertToPage(a, peri, plant, r))
```

In general, users check proof obligations by inspecting the VDM-SL model, though new Overture tools are being developed to check the majority of proof obligations automatically using formal proof and related techniques. You can also note in Figure 3.11 that in the *Proof Obligation Explorer* view there is a status field and in there a few of the proof obligations have a checkmark. This is used to indicate that these proof obligations are trivially satisfied. It is also possible to get rid of such proof obligations in the list by pressing the *Filter proved* button at the top of the *Proof Obligation Explorer* view.

3.8 A Command-Line Interface

The graphical user interface presented so far has been designed for a human interacting with Overture. In addition, a simple command-line interface allows automatic batch execution of tests and provides a full set of interactive execution and debugging commands which can be useful when examining batch tests. The command-line also provides access to tool facilities that have not yet been included in the Overture GUI.

The engine underpinning Overture, called VDMJ, is written in Java, and so to run it from the command line, the VDMJ jar file ⁶ should be executed with a Java JRE (version 5 or later):

```
java -jar vdmj-2.0.0.jar
```

If the jar file is executed with no further options like this, it will print a list of available options and exit. The most important option is the VDM dialect that the tool should use. In the case of our alarm example, we want to use VDM-SL for which the option is `-vdmsl`. After this, we can simply specify the names of the VDM-SL model files to load, or the name of a directory from which to load all VDM-SL model files:

```
java -jar vdmj-2.0.0.jar -vdmsl AlarmSL
```

That will perform a syntax and type check of all the VDM-SL model files in the `AlarmSL` directory, producing any errors and warning messages on the console, before terminating:

⁶See the Overture documentation at sourceforge.net/projects/overture for the location of the jar file.



```
Parsed 1 module in 0.391 secs. No syntax errors
Type checked 1 module in 0.063 secs. No type errors
```

In the case of our alarm example, there are no syntax or type checking errors. Any warnings can be suppressed using the `-w` option.

If a VDM-SL model has no type checking errors, it can either be given an expression to evaluate as an option on the command line, or the user can enter an interactive mode to evaluate expressions and debug their execution.

To evaluate an expression from the command line, the `-e` option is used, followed by a VDM expression to evaluate. You may also find the `-q` option useful, as this suppresses the informational messages about the parsing and type checking:

```
java -jar vdmj-2.0.0.jar -vdmsl -q -e
    "ExpertIsOnDuty(e1, p1)" AlarmSL
```

This produces a single line of output for the evaluation, since the parsing and checking messages are suppressed, and there are no warnings:

```
{mk_token("Monday day"), mk_token("Tuesday day")}
```

Clearly a batch of test evaluations could be performed automatically by running a series of similar commands and saving the output results for comparison against expected results.

To run the command line interpreter interactively, the `-i` command line option must be given. Instead of terminating after the type check, this will cause VDMJ to enter its interactive mode, and give the interactive `>` prompt:

```
Parsed 1 module in 0.391 secs. No syntax errors
Type checked 1 module in 0.047 secs. No type errors
Initialized 1 module in 0.063 secs.
Interpreter started
>
```

From this prompt, various interactive commands can be given to evaluate expressions, debug their evaluation, or examine the VDM-SL model environment. The `help` command lists the commands available. The `quit` command leaves the interpreter. For example, the following session illustrates the evaluation of a `Run` operation, and a debug session with a breakpoint at the start of the same operation:

```
> print Run(e1)
= {mk_token("Monday day"), mk_token("Tuesday day")}
Executed in 0.016 secs.
```



```
> break ExpertIsOnDuty
Created break [1] in 'DEFAULT' (AlarmSL\alarm.vdmsl)
at line 39:6
39:{peri| peri in set dom sch & ex in set sch(peri)};
> print Run(e1)
Stopped break [1] in 'DEFAULT' (AlarmSL\alarm.vdmsl)
at line 39:6
39:{peri| peri in set dom sch & ex in set sch(peri)};
[thread 1]> print sch
sch = {mk_token("Tuesday night") |->
  {mk_Expert(mk_token(174), {<Elec>, <Chem>,
    <Bio>, <Mech>}})},
  mk_token("Monday day") |->
  {mk_Expert(mk_token(181), {<Elec>, <Mech>}},
    mk_Expert(mk_token(169), {<Chem>, <Bio>}},
    mk_Expert(mk_token(134), {<Elec>}})},
  mk_token("Monday night") |->
  {mk_Expert(mk_token(174), {<Elec>, <Chem>,
    <Bio>, <Mech>}})},
  mk_token("Tuesday day") |->
  {mk_Expert(mk_token(134), {<Elec>}},
    mk_Expert(mk_token(154), {<Bio>, <Chem>, <Elec>}},
    mk_Expert(mk_token(190), {<Mech>, <Bio>}})}}
[thread 1]> continue
= {mk_token("Monday day"), mk_token("Tuesday day")}
Executed in 8.966 secs.
>
```

Notice that the `print` command is available at the breakpoint to examine the runtime state of the system. In the example, we print out the value of the `sch` variable. The `help` command is context sensitive, and will list the extra debugging commands available at a breakpoint, such as `continue`, `step`, `stack`, `list` and so on. The full set of commands is described in the VDMJ User Guide⁷.

3.9 Summary

We have introduced the following features of Overture:

- project setup of selected VDM-SL files;

⁷Supplied with the Overture documentation.



- syntax and type checking of VDM-SL models;
- error reporting;
- executing and debugging VDM-SL models;
- generating proof obligations for VDM-SL models; and
- using the command-line interface.

Exercise 3.3★ Imagine an extension to the alarm example which would enable experts to swap duties. This function is called `ChangeExpert`. Given a plant, two experts and a period it will yield a new plant where the plan has been changed so that the first expert will be replaced by the second expert in the given period. A first version of this function could be formulated as

```
ChangeExpert: Plant * Expert * Expert * Period -> Plant
ChangeExpert (mk_Plant (plan,alarms),ex1,ex2,peri) ==
  mk_Plant (plan ++ {peri |-> plan(peri)\{ex1} union {ex2}},
            alarms)
```

where the `\` symbol removes the `ex1` value from the schedule for the given period `peri` and **union** adds the `ex2` value.

Do you see any problems with this function? This definition is placed in the file `changeexpert.vdmsl` which is a part of the AlarmSL project. using this definition it is possible to debug expressions such as:

```
ChangeExpert (plant1,e4,e7,p3)
ChangeExpert (plant1,e3,e7,p3)
```

Will the invariant on the `Plant` data type be violated? Test this by setting the option for invariant checking. If the invariant is broken it is possible to make a break point for the invariant `inv_Plant` itself and call that with a `Plant` value which possibly satisfies the invariant. Single stepping inside this makes it easier to discover how the invariant is broken⁸. If necessary, add the pre-condition needed to complete the function. Try to generate the proof obligations for the `changeexpert.vdmsl` file and see if you can find the proof obligation ensuring that the invariant cannot be broken. □

⁸Note that in the current version of the Overture IDE violating such invariants an internal error (0029) will occur so to see what is going on it is advisable to put a breakpoint in the `ChangeExpert` function and then step into it, so you can see the evaluation of the invariant.



References

- [Fitzgerald&98] John Fitzgerald and Peter Gorm Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, 1998.



Appendix A

A Chemical Plant Example

This appendix presents the requirements for a simple alarm system for a chemical plant. It forms a running example that serves to illustrate the process described earlier and to introduce elements of the VDM++ modelling language. Although the modelling process is described here as though it were a single-pass activity, a real development would usually be iterative.

A.1 An informal description

The example is inspired by a subcomponent of a large alarm system developed by IFAD A/S and introduced in [Fitzgerald&98]. Chapter 3 provides an interactive and hands-on tour of the tools available for supporting the development of the model.

Imagine that you are developing a system that manages the calling out of experts to deal with operational faults discovered in a chemical plant. The plant is equipped with sensors that are able to raise alarms in response to conditions in the plant. When an alarm is raised, an expert must be called to the scene. Experts have different qualifications for coping with different kinds of alarms. It has been decided to produce a model to ensure that the rules concerning the duty schedule and the calling out of experts are correctly understood and implemented. The individual requirements are labelled R1, R8 for further reference:

- R1.** A computer-based system is to be developed to manage the alarms of this plant.
- R2.** Four kinds of qualifications are needed to cope with the alarms: electrical, mechanical, biological, and chemical.
- R3.** There must be experts on duty during all periods allocated in the system.
- R4.** Each expert can have a list of qualifications.
- R5.** Each alarm reported to the system has a qualification associated with it along with a description of the alarm that can be understood by the expert.



- R6.** Whenever an alarm is received by the system an expert with the right qualification should be found so that he or she can be paged.
- R7.** The experts should be able to use the system database to check when they will be on duty.
- R8.** It must be possible to assess the number of experts on duty.

In the next section the development of a model of an alarm system to meet these requirements is initiated. The purpose of the model is to clarify the rules governing the duty roster and calling out of experts to deal with alarms.

A.2 A VDM-SL model of the Alarm example

This section presents the full VDM-SL model of the alarm example. However, it does so without any explanatory text. That is placed in the VDM-SL book so if you are a newcomer to VDM-SL please read that there.

types

```
Plant :: schedule : Schedule
      alarms    : set of Alarm
inv mk_Plant(schedule,alarms) ==
    forall a in set alarms &
      forall peri in set dom schedule &
        QualificationOK(schedule(peri),a.quali);

Schedule = map Period to set of Expert
inv sch ==
  forall exs in set rng sch &
    exs <> {} and
    forall ex1, ex2 in set exs &
      ex1 <> ex2 => ex1.expertid <> ex2.expertid;

Period = token;

Expert :: expertid : ExpertId
       quali      : set of Qualification
inv ex == ex.quali <> {};

ExpertId = token;

Qualification = <Elec> | <Mech> | <Bio> | <Chem>;
```



```
Alarm :: alarmtext : seq of char
      quali       : Qualification
```

The functionality can be defined as follows.

functions

```
NumberOfExperts: Period * Plant -> nat
NumberOfExperts(peri, plant) ==
  card plant.schedule(peri)
pre peri in set dom plant.schedule;

ExpertIsOnDuty: Expert * Plant -> set of Period
ExpertIsOnDuty(ex, mk_Plant(sch, -)) ==
  {peri | peri in set dom sch & ex in set sch(peri)};

ExpertToPage(a:Alarm, peri:Period, plant:Plant) r: Expert
pre peri in set dom plant.schedule and
  a in set plant.alarms
post r in set plant.schedule(peri) and
  a.quali in set r.quali;

QualificationOK: set of Expert * Qualification -> bool
QualificationOK(exs, reqquali) ==
  exists ex in set exs & reqquali in set ex.quali
```

functions

```
ChangeExpert: Plant * Expert * Expert * Period -> Plant
ChangeExpert(mk_Plant(plan, alarms), ex1, ex2, peri) ==
  mk_Plant(plan ++ {peri |-> plan(peri)\{ex1} union {ex2}},
    alarms)
```

values

```
p1:Period = mk_token("Monday day");
p2:Period = mk_token("Monday night");
p3:Period = mk_token("Tuesday day");
p4:Period = mk_token("Tuesday night");
p5:Period = mk_token("Wednesday day");
```



```
eid1:ExpertId = mk_token(134);
eid2:ExpertId = mk_token(145);
eid3:ExpertId = mk_token(154);
eid4:ExpertId = mk_token(165);
eid5:ExpertId = mk_token(169);
eid6:ExpertId = mk_token(174);
eid7:ExpertId = mk_token(181);
eid8:ExpertId = mk_token(190);

e1:Expert = mk_Expert(eid1,{<Elec>});
e2:Expert = mk_Expert(eid2,{<Mech>,<Chem>});
e3:Expert = mk_Expert(eid3,{<Bio>,<Chem>,<Elec>});
e4:Expert = mk_Expert(eid4,{<Bio>});
e5:Expert = mk_Expert(eid5,{<Chem>,<Bio>});
e6:Expert = mk_Expert(eid6,{<Elec>,<Chem>,<Bio>,<Mech>});
e7:Expert = mk_Expert(eid7,{<Elec>,<Mech>});
e8:Expert = mk_Expert(eid8,{<Mech>,<Bio>});

s: map Period to set of Expert
  = {p1 |-> {e7,e5,e1},
     p2 |-> {e6},
     p3 |-> {e1,e3,e8},
     p4 |-> {e6}};

a1:Alarm = mk_Alarm("Power supply missing",<Elec>);
a2:Alarm = mk_Alarm("Tank overflow",<Mech>);
a3:Alarm = mk_Alarm("CO2 detected",<Chem>);
a4:Alarm = mk_Alarm("Biological attack",<Bio>);

plant1: Plant = mk_Plant(s,{a1,a2,a3,a4})
```