Overture – Open-source Tools for Formal Modelling TR-2009-02
December 2009

User Guide for the Overture VDM Tool Support

by

Peter Gorm Larsen, Kenneth Lausdahl, Augusto Reibero 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

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 enable 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 [Bjørner&78a, Jones90, Fitzgerald&08a]. It consists of a group of mathematically well-founded languages for expressing system models during early design stages, before expensive implementation commitments are made. The construction and analysis of 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 [Larsen&95, Clement&99, Kurita&09]. 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 are expressed in a specification language (VDM-SL) that supports the description of data and functionality [ISOVDM96, Fitzgerald&98, Fitzgerald&09]. Data are defined by means of types built using constructors that define structured data and collections such as sets, sequences and mappings from basic values such as Booleans and 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 [Fitzgerald&05]. An additional extension to VDM++ is called VDM Real Time (VDM-RT) (formerly called VDM In a Constrained Environment (VICE)) [Mukherjee&00, Verhoef&06]. 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 [Larsen&10]. It can be seen as an open source version of the commercial tool called VDMTools [Elmstrøm&94, Fitzgerald&08b] although that also have features to generate executable code in high-level programming languages which 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 explanantion about how to get hold of the software in Ssection 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 explainantion 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 specfic 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 to and from UML models is presented. In Section 12 it is illustrated how one can move from a VDM++ project to a new VDM-RT project. In Section 13 it is shown how support to analysing and displaying logs from executing such VDM-RT models. After these sections the main part of the user manual is completed in Section 14 with an explanantion of the features from Overture that also is available from a command-line interface. Finally in Appendix **??** an index of significant terms used in this user manual can be found.

2 Getting Hold of the Software

The Overture project is managed on 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/files/
```

you can find pre-installed versions of Overture for Windows, Linux and Mac. At a later stage it will also be possible to use an update site to install it from directly in Eclipse. However, at the moment only stand-alone versions are distributed because the risk of version problems and dependencies with other plug-ins is much smaller this way.

Zip files with a large collection of existing VDM-SL, VDM++ and VDM-RT projects can be downloaded from

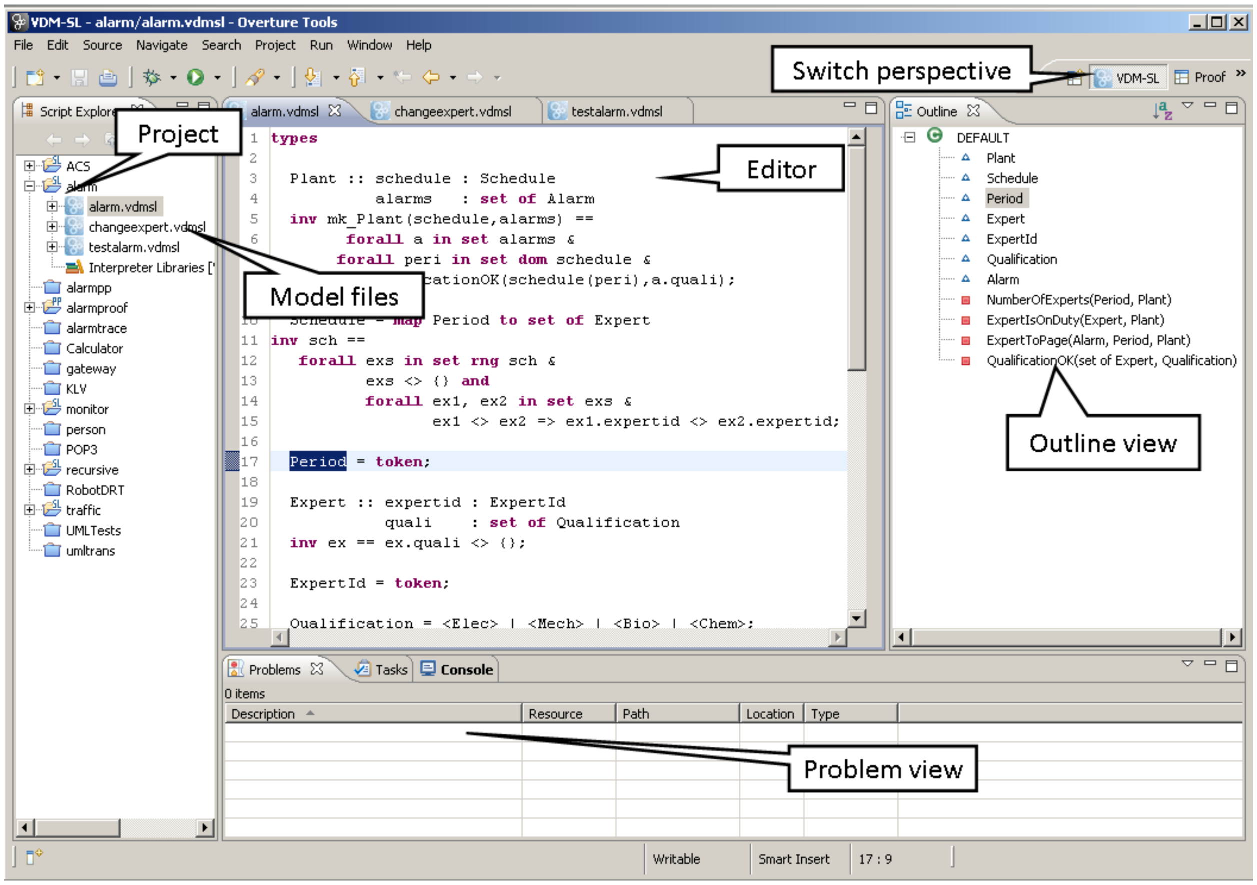
```
http://sourceforge.net/projects/overture/files/Examples/
```

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

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 about a perspective as a useful composition of views for conducting a particular task.



```
types

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

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.

Depending upon the dialect of VDM used in a given project, a corresponding Overture editor will be available here. A new VDM-SL project is created choosing the *File* → *New* → *Project*. Then Figure 2 will appear and *Next* can be used and then a name needs to be given to the project.

The *Outline view*, to the right of the editor (see Figure 3), presents an outline of the file selected in the editor. The outline 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 1 shows the outline view on the right hand side. 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 optionally sort what is displayed in the outline view, for instance it is possible to hide variables.

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

Most of the other features of the workbench, such as the menus and toolbars, are similar to

Figure 2: Creating a New VDM-SL Project

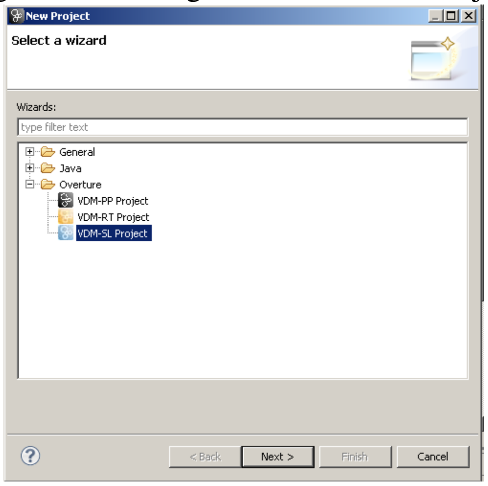
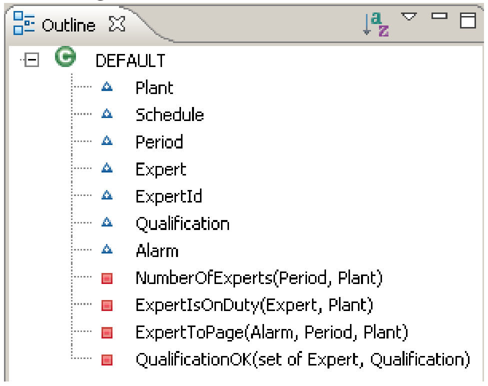


Figure 3: The Outline View


Outline
DEFAULT
Plant
Schedule
Period
Expert
ExpertId
Qualification
Alarm
NumberOfExperts(Period, Plant)
ExpertIsOnDuty(Expert, Plant)
ExpertToPage(Alarm, Period, Plant)
QualificationOK(set of Expert, Qualification)

other Eclipse applications, though note that there is 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 Applicabe in Overture

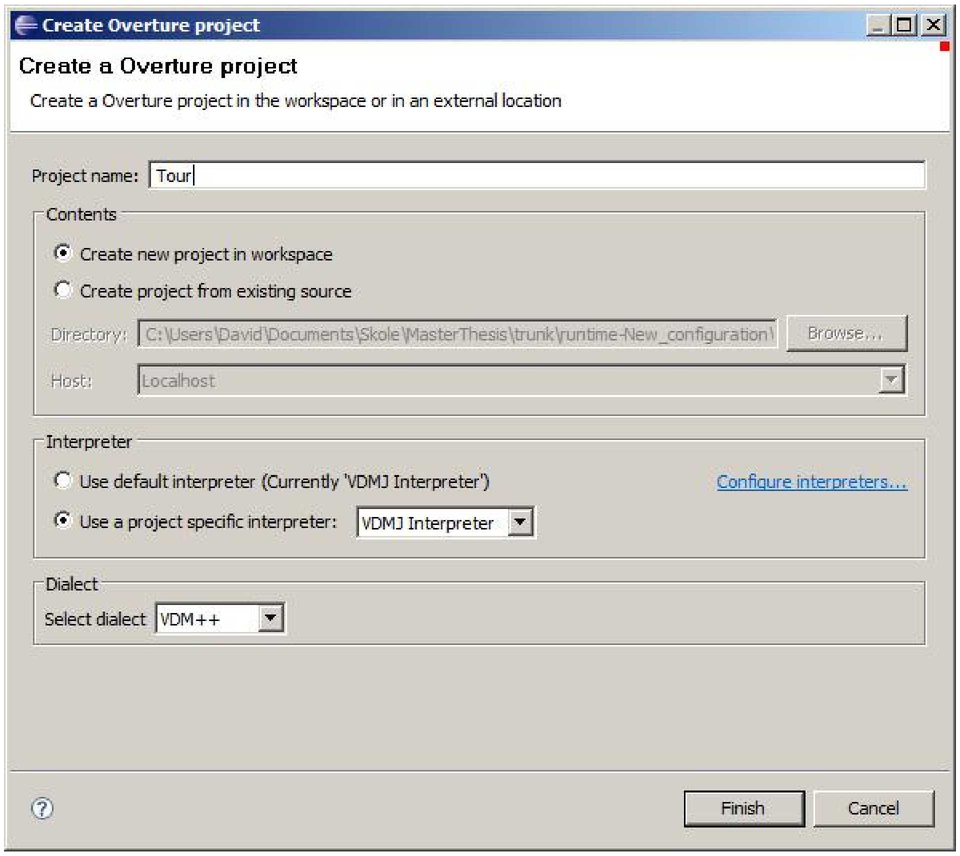
4 Managing Overture Projects

4.1 Importing Overture Projects

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. Type in a project name
4. Chose whether you would like the contents of the new project to be in your workspace or outside from existing source files and
5. click the finish button (see 4).


Create Overture project

Create a Overture project

Create a Overture project in the workspace or in an external location

Project name: Tour

Contents

(●) Create new project in workspace

( ) Create project from existing source

Directory: C:\Users\David\Documents\Skole\MasterThesis\trunk\runtime-New_configuration\

Host: Localhost

Interpreter

( ) Use default interpreter (Currently 'VDMJ Interpreter')

(●) Use a project specific interpreter: VDMJ Interpreter

Dialect

Select dialect VDM++

Figure 4: Create Project Wizard

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 for the chosen Overture project. Then a new file with the appropriate file extension according to the chosen dialect will 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 for kinds of definitions that will not be used can be deleted.

4.4 Setting Project Options

5 Editing VDM models

6 Interpretation and Debugging in Overture

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

6.1 Debug configuration

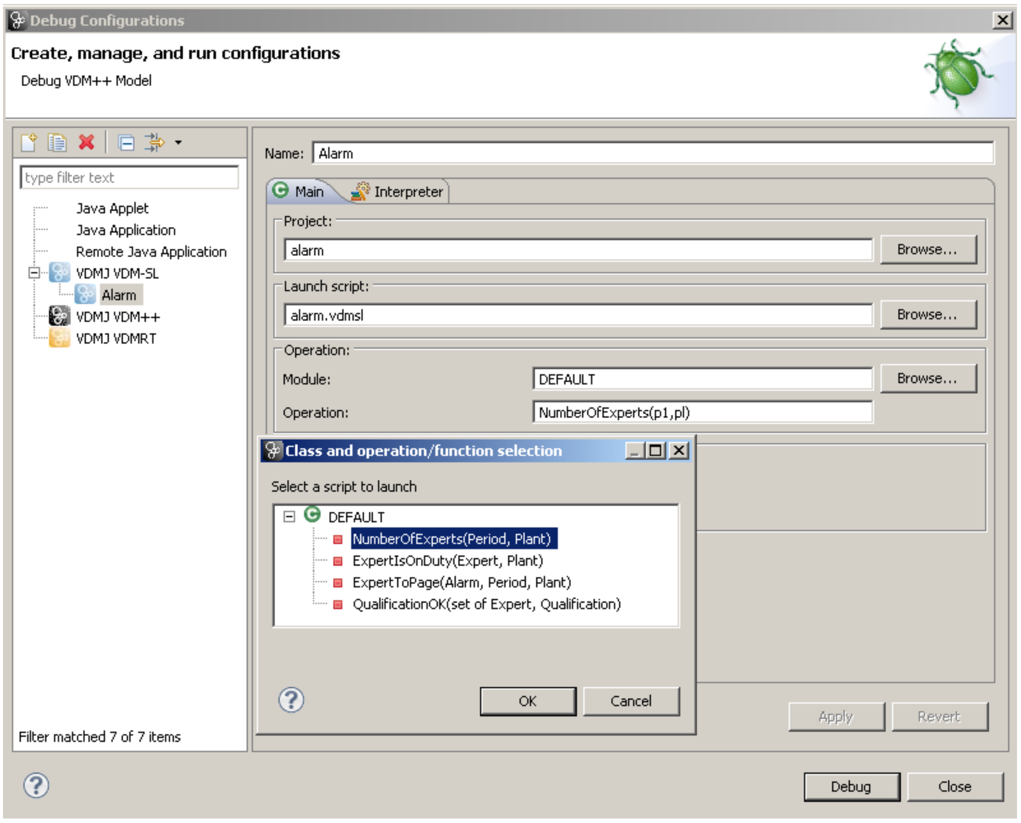
Debugging the model under development is done by creating a debug configuration from the menu *Run → Debug configuration …* The debug configuration dialog requires the following information as input to start the debugger: the project name, the class, the starting operation/function and the file containing the starting operation/function. Figure 5 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.

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 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 provided in the debug perspective are described below. The Debug Perspective is illustrated on figure 6

The *Debug view* is located in the upper left corner in the Debug perspective. The Debug view shows all running models and the call stacks belonging to them. It also shows whether a given



Name: Alarm

Project: alarm

Launch script: alarm.vdmsl

Operation:
Module: DEFAULT
Operation: NumberOfExperts(p1,pl)

Select a script to launch

DEFAULT
- NumberOfExperts(Period, Plant)
- ExpertIsOnDuty(Expert, Plant)
- ExpertToPage(Alarm, Period, Plant)
- QualificationOK(set of Expert, Qualification)

Figure 5: The debug configuration dialog

model is stopped, suspended or running. All threads are also shown, along with their running status. It is possible to switch between threads from the Debug view.

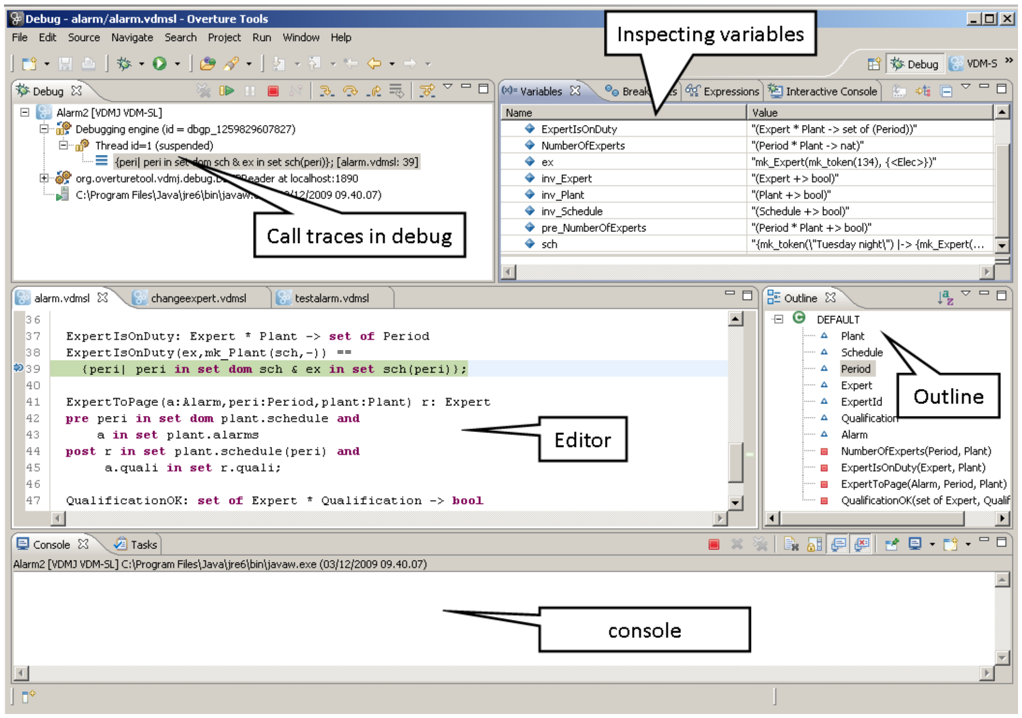
At the top of the view are buttons for controlling debugging such as; stop, step into, step over and resume. These are standard Eclipse debugging buttons (see Table 1).

6.2.1 Debug View

The debug View is located in the upper left corner in the Debug Perspective - see figure 6. 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.

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



```
Alarm2 [VDMJ VDM-SL]
  Debugging engine (id = dbgp_1259829607827)
    Thread id=1 (suspended)
      {peri| peri in set dom sch & ex in set sch(peri)}; [alarm.vdmsl: 39]
  org.overturetool.vdmj.debug.D…Reader at localhost:1890
  C:\Program Files\Java\jre6\bin\javaw.…/12/2009 09.40.07)
```

Name	Value
ExpertIsOnDuty	"(Expert * Plant -> set of (Period))"
NumberOfExperts	"(Period * Plant -> nat)"
ex	"mk_Expert(mk_token(134), {<Elec>})"
inv_Expert	"(Expert +> bool)"
inv_Plant	"(Plant +> bool)"
inv_Schedule	"(Schedule +> bool)"
pre_NumberOfExperts	"(Period * Plant +> bool)"
sch	"{mk_token(\"Tuesday night\") |-> {mk_Expert(…

```

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
```

```
DEFAULT
  Plant
  Schedule
  Period
  Expert
  ExpertId
  Qualification
  Alarm
  NumberOfExperts(Period, Plant)
  ExpertIsOnDuty(Expert, Plant)
  ExpertToPage(Alarm, Period, Plant)
  QualificationOK(set of Expert, Qualif
```

```
Alarm2 [VDMJ VDM-SL] C:\Program Files\Java\jre6\bin\javaw.exe (03/12/2009 09.40.07)
```

Figure 6: Debugging perspective

Table 1: Overture debugging buttons

Button	Explanation
▮▶	Resume debugging
▮▮	Suspend debugging
■	Terminate debugging
↳	Step into
↷	Step over
↱	Step return
⤵	Use step filters

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 arrays and so forth.

6.2.3 Breakpoints View

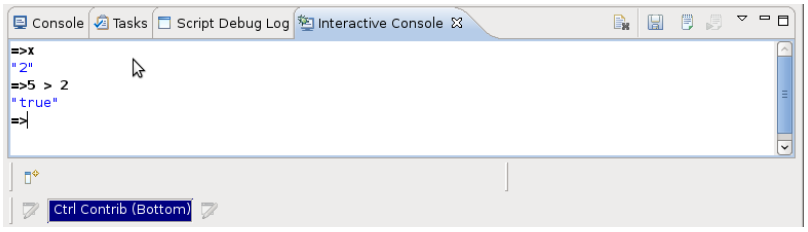
Breakpoints can be added both from the edit perspective and the debug perspective from the editor view. In the debug perspective however, there is a breakpoints view that shows all breakpoints. From the breakpoints view the user can easily navigate to the location of a given breakpoint, disable, delete or set the hit count or a break condition. In figure 6 the Breakpoints View is hidden behind the Variables View in the upper right hand corner in a tabbed notebook. Section 6.2.6 explains how to use conditional breakpoints.

6.2.4 Expressions View

The expressions view allows the user to write expressions, as for the variables view, the expressions are automatically updated when stepping. 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.

6.2.5 Interactive Console View

While the Expressions View allows to easily inspect values, the functionality is somewhat limited compared with 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 7 shows the interactive console.



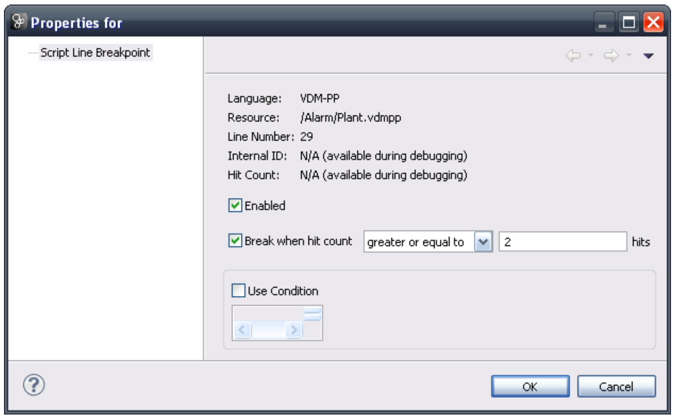
```
=>x
"2"
=>5 > 2
"true"
=>
```

Figure 7: The interactive console

6.2.6 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 8. It is possible to choose between two different conditional breakpoints, a hit count condition and one based on an expression defined by the user.



Language: VDM-PP
Resource: /Alarm/Plant.vdmpp
Line Number: 29
Internal ID: N/A (available during debugging)
Hit Count: N/A (available during debugging)

[✓] Enabled

[✓] Break when hit count greater or equal to 2 hits

[ ] Use Condition

Figure 8: Conditional breakpoint options

7 Collecting Test Coverage Information

8 Pretty Printing to LaTeX

9 Managing Proof Obligations

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 not yet available for VDM-SL models). 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 test cases that have detected errors in the VDM++ model can easily be found and 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`)[1].

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, ‘)’ ;

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

Trace apply expressions are the most basic element in trace definitions. The identifier before the "`.`" indicate an object for with 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

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.

ℹ**:** 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

12 Moving from VDM++ to VDM-RT

In the methodology for the development of distributed real-time embedded systems using the VDM techniology there is a step where one moves from a VDM++ model to a VDM-RT model [Larsen&09]. 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. In the menu that comes up one then need to select the *Overture Utility* → *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 an easy way to fast get the starting point for a VDM-RT model developed. 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 RealTime 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 9 the architectural overview of the system is given, describing the distributed nature of the model.

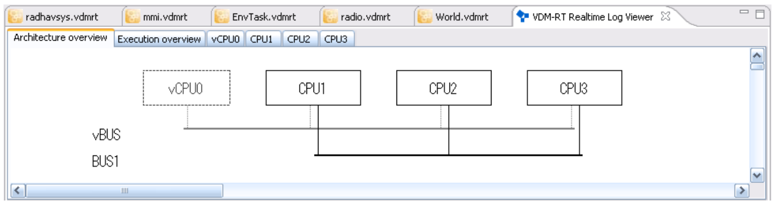


Figure 9: Architectural overview

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 10. This view shows how the different CPUs communicate via the BUSes of the system.

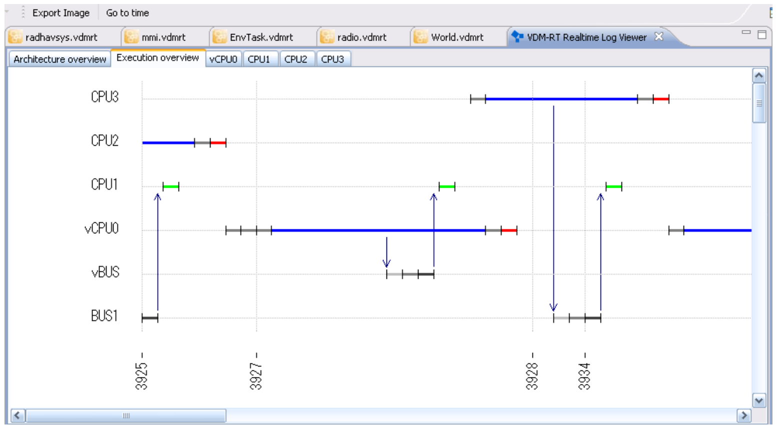


Figure 10: Execution overview

Since the complete execution of the model cannot be shown in a normal sized window, the user has the option of jumping the 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 11.

14 A Command-Line Interface to VDMJ

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

14.1 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>]
-vdmsl: parse files as VDM-SL
-vdmpp: parse files as VDM++
```

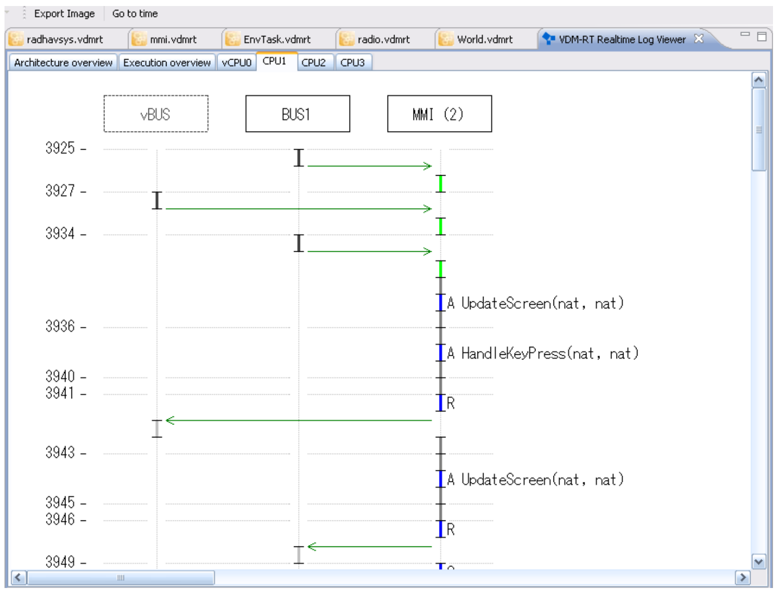


Figure 11: Execution on single CPU

```
-vdmrt: parse files as VICE
-w: suppress warning messages
-q: suppress information messages
-i: run the interpreter if successfully type checked
-p: generate proof obligations and stop
-e <exp>: evaluate <exp> and stop
-c <charset>: select a file charset
-t <charset>: select a console charset
-o <filename>: saved type checked specification
-pre: disable precondition checks
-post: disable postcondition checks
-inv: disable type/state invariant checks
-dtc: disable all dynamic type checking
-log: enable real-time event logging
```

Notice that the error indicates that the tool must be invoked with either the `-vdmsl`, `-vdmpp` or `-vdmrt` option to indicate the VDM dialect and parser required.

Normally, a specification 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 no `-i` option is given, the tool will parse and type check the specification files only, giving any errors and warnings on standard output, then stop. Warnings can be suppressed with the `-w` option. The `-q` option can be used to suppress the various information messages printed (this does not include errors and warnings).

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.

The -c and -t options allow the file and console character sets to be defined, respectively. This is to allow a specification written in languages other than the default for your system to be used (see section 3).

The -o option allows a parsed and type checked specification to be saved to a file. Such files are effectively libraries, and can be can be re-loaded without the parsing/checking overhead. The -pre, -post, -inv and -dtc options can be used to disable precondition, postcondition, invariant and dynamic type checking, respectively. By default, all these checks are performed. The -log option is for use with -vdmrt, and causes real-time events from the model to be written to the file name given. These are useful with the Overture Eclipse GUI, which has a plugin to display timing diagrams [8].

If flat.vdmsl contains a simple VDM-SL specification of the factorial function, called "fac", the following illustrate ways to test the specification, with user input shown in bold:

```
functions
fac: int -> int
fac(a) == if a < 2 then 1 else a * fac(a-1)
pre a > 0
```

```
$ java -jar vdmj-2.0.0.jar -vdmsl flat.vdmsl
Parsed 1 module in 0.202 secs. No syntax errors
Warning 5012: Recursive function has no measure in (flat.vdmsl) at
line 3:1
Type checked 1 module in 0.016 secs. No type errors and 1 warning
```

```
$ java -jar vdmj-2.0.0.jar -vdmsl -q -w flat.vdmsl
<quiet>
```

```
$ java -jar vdmj-2.0.0.jar -vdmsl -w -e "fac(10)" flat.vdmsl
Parsed 1 module in 0.28 secs. No syntax errors
Type checked 1 module in 0.031 secs. No type errors,
suppressed 1 warning
Initialized 1 module in 0.031 secs.
3628800
Bye
```

```
$ java -jar vdmj-2.0.0.jar -vdmsl -e "fac(10)" -q -w flat.vdmsl
3628800
```

```
$ java -jar vdmj-2.0.0.jar -vdmsl -i -w flat.vdmsl
Parsed 1 module in 0.202 secs. No syntax errors
Type checked 1 module in 0.016 secs. No type errors,
suppressed 1 warning
Initialized 1 module in 0.031 secs.
Interpreter started
> print fac(10)
= 3628800
Executed in 0.0 secs.
> quit
Bye
```

```
$ java -jar vdmj-2.0.0.jar -vdmsl -p -w flat.vdmsl
Parsed 1 module in 0.218 secs. No syntax errors
Type checked 1 module in 0.015 secs. No type errors,
suppressed 1 warning
Generated 1 proof obligation:
Proof Obligation 1:
fac: function apply obligation in 'DEFAULT1' (flat.vdmsl) at
line 4:38
(forall a:int & (a > 0) =>
(not (a < 2) =>
pre_f((a - 1))))
```

```
$ java -jar vdmj-2.0.0.jar -vdmsl -w -e "fac(0)" -w flat.vdmsl
Parsed 1 module in 0.203 secs. No syntax errors
Type checked 1 module in 0.015 secs. No type errors,
suppressed 1 warning
Initialized 1 module in 0.016 secs.
Execution: Error 4055: Precondition failure: pre_f in (flat.vdmsl)
at line 5:11
   a = 0
   fac = (int -> int)
   pre_fac = (int +> bool)
In root context of fac(a) in 'DEFAULT1' (console) at line 1:1
In root context of interpreter in 'DEFAULT1' (flat.vdmsl) at
line 3:1
In root context of global environment
Bye
```

```
$ java -jar vdmj-2.0.0.jar -vdmsl -w -pre -e "fac(0)"-w flat.vdmsl
```

```
Parsed 1 module in 0.218 secs. No syntax errors
Type checked 1 module in 0.016 secs. No type errors,
suppressed 1 warning
Initialized 1 module in 0.015 secs.
1
Bye
```

```
$ java -jar vdmj-2.0.0.jar -vdmsl -w -o flat.lib flat.vdmsl
Parsed 1 module in 0.203 secs. No syntax errors
Type checked 1 module in 0.016 secs. No type errors,
suppressed 1 warning
Saved 1 module to flat.lib in 0.093 secs.
```

```
$ java -jar vdmj-2.0.0.jar -vdmsl flat.lib -e "fac(10)"
Loaded 1 module from flat.lib in 0.187 secs
Initialized 1 module in 0.0 secs.
3628800
Bye
```

14.2 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; the markup is ignored by the parser, though reported line numbers will be correct.

The Java program 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.

14.3 The Interpreter

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. The interpreter prompt is ">". The following illustrates some of the interactive interpreter commands (explanation below. The shmem source model is in Appendix A):

```
$ 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
```

```
> help
modules - list the loaded module names
default <module> - set the default module name
state - show the default module state
print <expression> - evaluate expression
assert <file> - run assertions from a file
init - re-initialize the global environment
env - list the global symbols in the default environment
pog - generate a list of proof obligations
break [<file>:]<line#> [<condition>] - create a breakpoint
break <function/operation> [<condition>] - create a breakpoint
trace [<file>:]<line#> [<exp>] - create a tracepoint
trace <function/operation> [<exp>] - create a tracepoint
remove <breakpoint#> - remove a trace/breakpoint
list - list breakpoints
coverage [<file>|clear] - display/clear file line coverage
latex|latexdoc [<files>] - generate LaTeX line coverage files
files - list files in the current specification
reload - reload the current specification files
load <files> - replace current loaded specification files
quit - leave the interpreter
```

```
> modules
M (default)
```

```
> state
M`Q4 = [mk_M(<FREE>, 0, 9999)]
M`rseed = 87654321
M`Memory = mk_Memory(87654321, [mk_M(<FREE>, 0, 9999)],
                               [mk_M(<FREE>, 0, 9999)])
M`Q3 = [mk_M(<FREE>, 0, 9999)]
```

```
> print rand(100)
= 71
Executed in 0.0 secs.
```

```
> print rand(100)
= 44
Executed in 0.0 secs.
```

```
> state
M`Q4 = [mk_M(<FREE>, 0, 9999)]
M`rseed = 566044643
M`Memory = mk_Memory(566044643, [mk_M(<FREE>, 0, 9999)],
                                [mk_M(<FREE>, 0, 9999)])
M`Q3 = [mk_M(<FREE>, 0, 9999)]
```

```
> init
Global context initialized
```

```
> state
M`Q4 = [mk_M(<FREE>, 0, 9999)]
M`rseed = 87654321
M`Memory = mk_Memory(87654321, [mk_M(<FREE>, 0, 9999)],
                               [mk_M(<FREE>, 0, 9999)])
M`Q3 = [mk_M(<FREE>, 0, 9999)]
```

```
> print rand(100)
= 71
Executed in 0.0 secs.
```

```
> print rand(100)
= 44
Executed in 0.0 secs.
```

```
> env
M`fragments = (M`Quadrant -> nat)
M`combine = (M`Quadrant -> M`Quadrant)
M`tryBest = (nat ==> nat)
M`seed = (nat1 ==> ())
M`reset = (() ==> ())
M`bestfit = (nat1 * M`Quadrant -> nat1)
M`add = (nat1 * nat1 * M`Quadrant -> M`Quadrant)
M`firstFit = (nat1 ==> bool)
M`rand = (nat1 ==> nat1)
M`tryFirst = (nat ==> nat)
M`main = (nat1 * nat1 ==> seq of (<SAME> | <BEST> | <FIRST>))
M`MAXMEM = 10000
M`delete = (M`M * M`Quadrant -> M`Quadrant)
M`inv_M = (M`M +> bool)
```

```
M`CHUNK = 100
M`bestFit = (nat1 ==> bool)
M`least = (nat1 * nat1 -> nat1)
M`fits = (nat1 * M`Quadrant -> nat1)
M`init_Memory = (M`Memory +> bool)
M`pre_add = (nat1 * nat1 * M`Quadrant +> bool)
```

```
> pog
Generated 36 proof obligations:
Proof Obligation 1:
M`fits: cases exhaustive obligation in 'M' (shmem.vdm) at
line 40:5
(forall size:nat1, Q:Quadrant &
Q = [] or Q = [h] ^ tail)
...
Proof Obligation 35:
M`tryBest: sequence apply obligation in 'M' (shmem.vdm) at
line 176:27
rand((len Q4)) in set inds Q4
Proof Obligation 36:
M`tryBest: subtype obligation in 'M' (shmem.vdm) at line 166:1
RESULT >= 0
```

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

The modules command lists the names of the modules loaded from the specification. In this example there is only one, called "M". One of the modules is identified as the default; names in the default module do not need to be qualified (so you can say print xyz rather than print M'xyz). The default module can be changed with the default command.

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

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

The assert command (illustrated below) can take a list of assertions from a file, and execute each of them in turn, raising an error for any assertion which is false. The assertions in the file must be simple boolean expressions, one per line:

References

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

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

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

An informal introduction to *Meta-IV*

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

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

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

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

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

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

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

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

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

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

One of several examples of ALGOL 60 descriptions.

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

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

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

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

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

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

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

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

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

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

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

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

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

Index