User Guide for the Overture Plug-in for Eclipse

David Holst Møller and Christian Thillemann

2009

Versions:

date	Version	Comment
03.06.2009	0.0.1	First version of the user guide. Overture IDE

Contents

Chapter 1

User Guide

This guide explains how to use the Overture IDE for developing models for VDM language. The two first section describe how to install the Overture IDE and create a first project. Next the functionality of the editor is explained. And lastly the debugger functionality of the debugger is described.

1.1 Installation

1.1.1 Installing Eclipse

The Overture IDE is build upon the Eclipse platform, which is an open source development platform. Eclipse can be downloaded from from the the web-site http://www.eclipse.org/downloads/.

1.1.2 Installing the Overture IDE

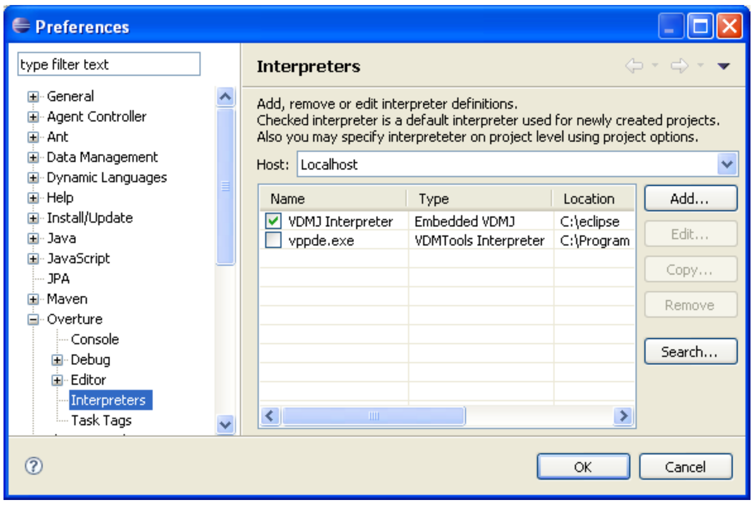
The Overture IDE plug-in is installed by following these steps:

1. Help → Software updates
2. Choose the tab “Available Software”
3. Click the button “Add Site...” and paste the following URL:
   http://www.overturetool.org/updatesite/
4. Choose Overture Editor and Dynamic Language Toolkit
5. Click the button Install and follow the installation wizard.
6. Restart Eclipse

1.2 Getting started

1.2.1 Defining interpreters

At the moment two different interpreter is supported by the Overture IDE, VDMJ and VDMTools. The VDMJ interpreter is bundled with the Overture IDE, but a special license is needed, if one wishes to make use of a VDMTools interpreter. Specifying the default interpreter is done in preferences(see figure 1.1), by selecting the menu Window → preferences → Overture → Interpreters.



Interpreters

Add, remove or edit interpreter definitions.
Checked interpreter is a default interpreter used for newly created projects.
Also you may specify interpreteter on project level using project options.

Host: Localhost

Name	Type	Location
✓ VDMJ Interpreter	Embedded VDMJ	C:\eclipse
vppde.exe	VDMTools Interpreter	C:\Program

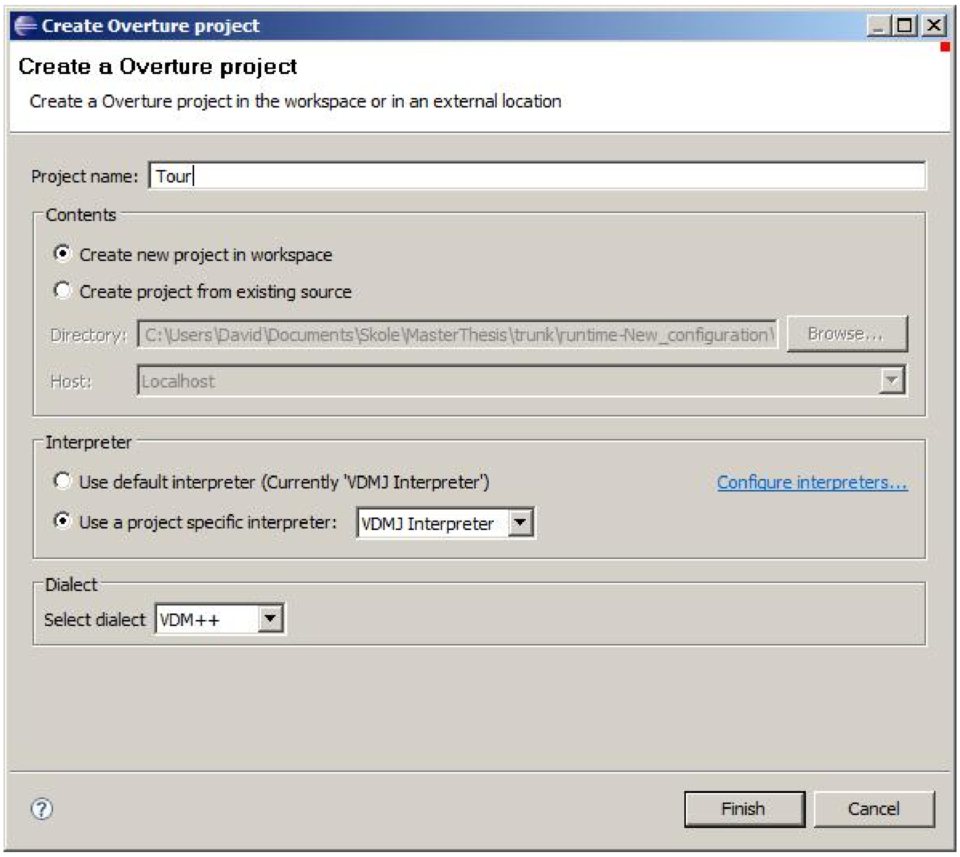
Figure 1.1: Choosing interpreter

If one wishes to use VDMJ just enable the 'check box' at VDMJ, for the VDMTools interpreter click the add. . . button and choose the VDMTools interpreter and the path to the executable file to VDMTools.

1.2.2 Creating a New Overture Project

1. Create a new project by choosing File → new → Overture Project
2. Type in a project name
3. Select dialect type: **VDM++**
4. Chose an interpreter type **VDMJ** or **VDMTools** and click the finish button see 1.2

Note: At the moment It doesn't matter which dialect and interpreter that is chosen. The dialect VDM++ will always be used.



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\ Browse...

Host: Localhost

Interpreter

Use default interpreter (Currently 'VDMJ Interpreter') Configure interpreters...

[✓ Use a project specific interpreter:] VDMJ Interpreter

Dialect

Select dialect VDM++

Finish Cancel

Figure 1.2: Create Project Wizard

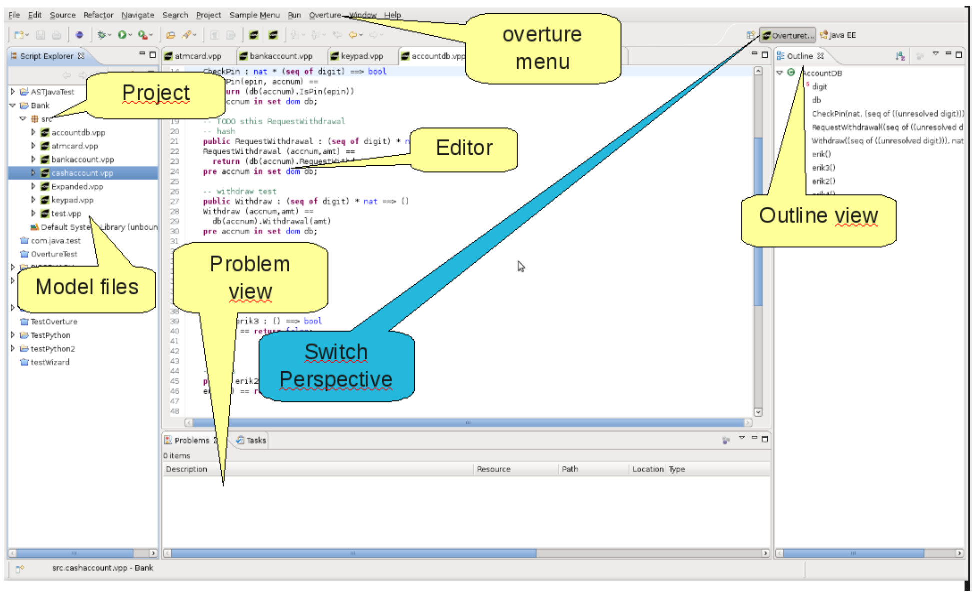
1.2.3 Creating Files

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 File
2. Right click on the overture project choose new → Overture File

1.3 Overture Perspective

Figure 1.3 shows the different views available when editing a model. Each view will be explained in this section.



```
CheckPin : nat * (seq of digit) ==> bool
  Pin(epin, accnum) ==
  urn (db(accnum).IsPin(epin))
  ccnum in set dom db;

-- TODO sthis RequestWithdrawal
-- hash
public RequestWithdrawal : (seq of digit) * n
RequestWithdrawal (accnum,amt) ==
  return (db(accnum).RequestWith
pre accnum in set dom db;

-- withdraw test
public Withdraw : (seq of digit) * nat ==> ()
Withdraw (accnum,amt) ==
  db(accnum).Withdrawal(amt)
pre accnum in set dom db;
```

Figure 1.3: The Overture Perspective

1.3.1 Project Explorer View

The Project Explorer View basically provides an overview of projects and files contained by these projects. double clicking on a model file will open the file in the Editor.

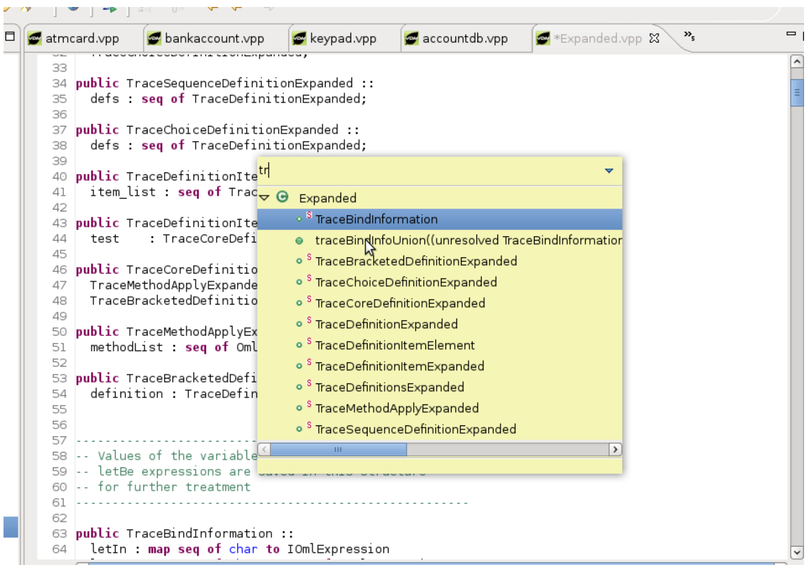
1.3.2 Overture Editor

The centre of figure 1.3 shows the Overture editor, with syntax highlighting. It is possible to change the colour of the highlighted words, which are divided into groups such as keywords, types, strings and comments, etc, in the preference page (window → preference → Overture → Editor → Syntax coloring).

The Overture IDE support intellisense, but it is only possible to complete keywords at the time of writing. The Overture IDE triggers the proposal when the user hits the key combination *CTRL+space*.

1.3.3 Outline

The outline view enables the user to navigate quickly in the model, figure 1.3 shows the outline view at the right side. A click on a operation/function will move the cursor in the editor to the definition of the operation. At the top of the outline view there is a number of buttons which gives the user the option to filter what is displayed in the outline view, for instance it is possible to hide variables.


atmcard.vpp
bankaccount.vpp
keypad.vpp
accountdb.vpp
*Expanded.vpp
tr
Expanded
TraceBindInformation
traceBindInfoUnion((unresolved TraceBindInformatior
TraceBracketedDefinitionExpanded
TraceChoiceDefinitionExpanded
TraceCoreDefinitionExpanded
TraceDefinitionExpanded
TraceDefinitionItemElement
TraceDefinitionItemExpanded
TraceDefinitionsExpanded
TraceMethodApplyExpanded
TraceSequenceDefinitionExpanded

```

public TraceSequenceDefinitionExpanded ::
  defs : seq of TraceDefinitionExpanded;

public TraceChoiceDefinitionExpanded ::
  defs : seq of TraceDefinitionExpanded;

public TraceDefinitionIte
  item_list : seq of Trac

public TraceDefinitionIte
  test    : TraceCoreDefi

public TraceCoreDefinitio
  TraceMethodApplyExpande
  TraceBracketedDefinitio

public TraceMethodApplyEx
  methodList : seq of Oml

public TraceBracketedDefi
  definition : TraceDefin


-------------------------
-- Values of the variable
-- letBe expressions are
-- for further treatment
-----------------------------------------------------

public TraceBindInformation ::
  letIn : map seq of char to IOmlExpression
```

Figure 1.4: Quick Outline

When editing a model, it can be useful to navigate to a specific operation in the file, instead of reaching for the mouse, this can be done using a shortcut (Ctrl+O)) which opens a small pop-up. This pop-up allows the user to navigate to a given location in the model simply by filling out the name of an field, operation, function or type. The use of intellisense furthermore ensures that the user does not have to write out the full name of the desired location. Figure 1.4 shows the quick outline.

1.3.4 Problems View

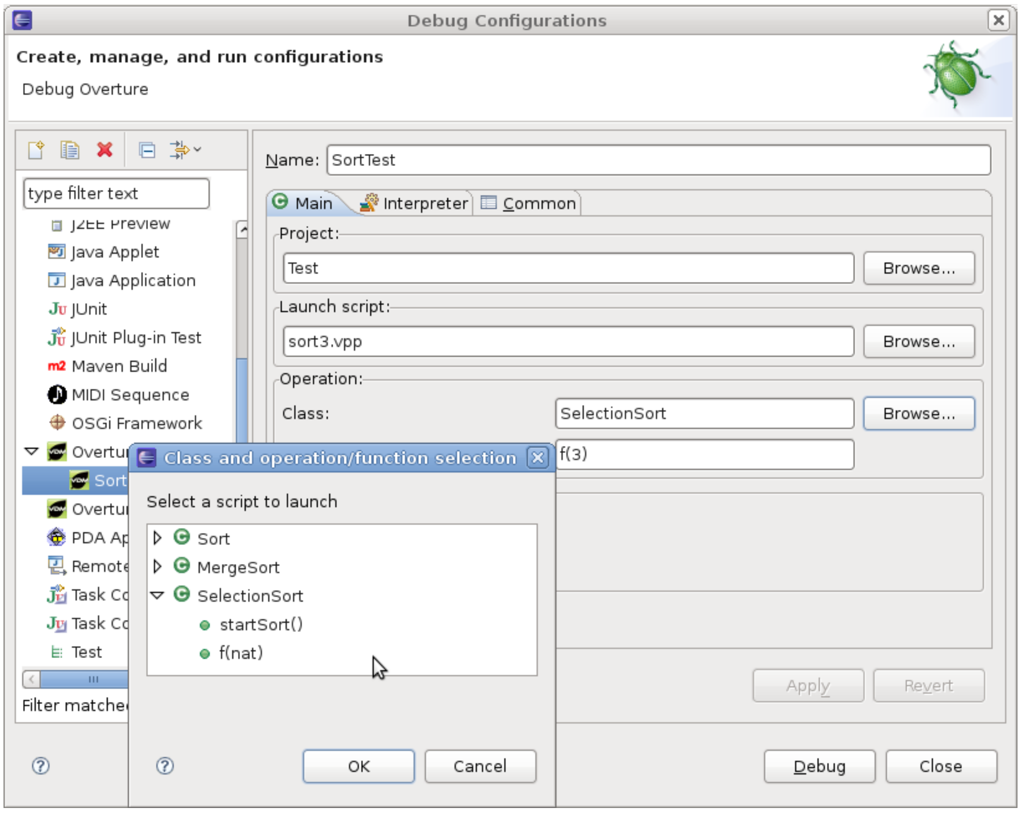
When editing a specification the Overture IDE will parse the content of the file, if there are any parse errors, it will be reported in the problems view. See bottom of figure 1.3. Each time a specification is saved the editor type checks the model and reports any errors or warnings.

1.4 Debugging

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

1.4.1 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 1.5 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.



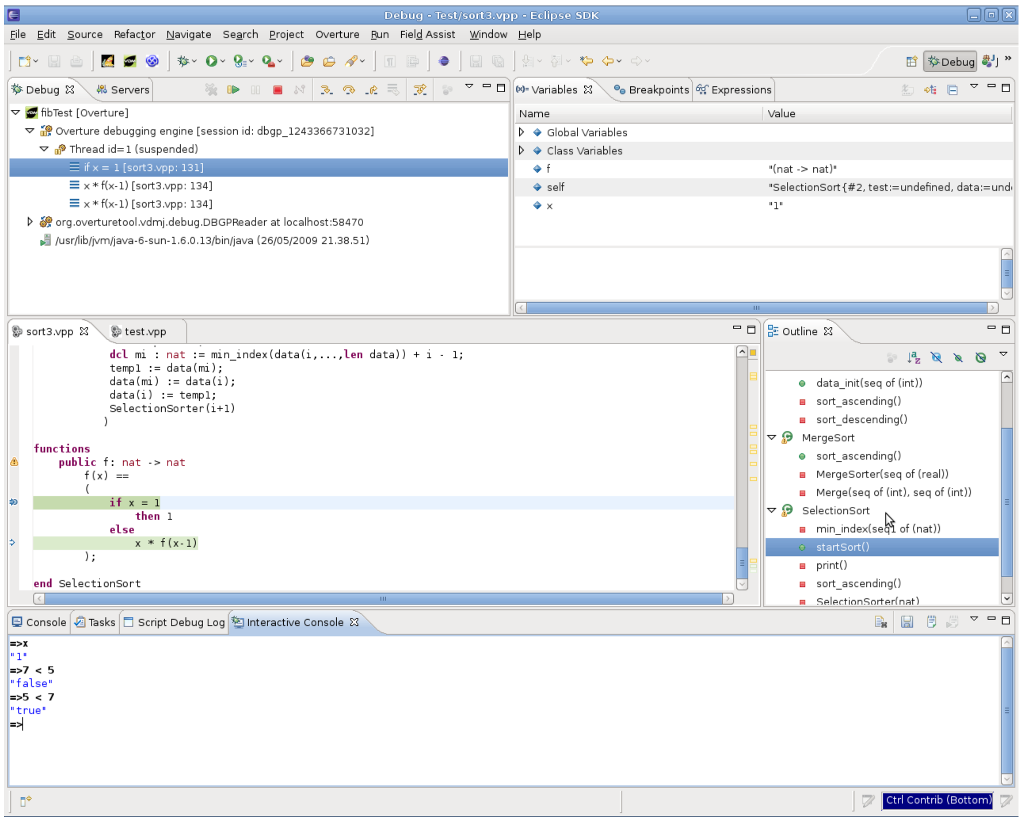
Name: SortTest
Project: Test
Launch script: sort3.vpp
Operation:
Class: SelectionSort
f(3)

Figure 1.5: The debug configuration dialog

1.4.2 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 1.6



```
fibTest [Overture]
  Overture debugging engine [session id: dbgp_1243366731032]
    Thread id=1 (suspended)
      if x = 1 [sort3.vpp: 131]
      x * f(x-1) [sort3.vpp: 134]
      x * f(x-1) [sort3.vpp: 134]
  org.overturetool.vdmj.debug.DBGPReader at localhost:58470
  /usr/lib/jvm/java-6-sun-1.6.0.13/bin/java (26/05/2009 21.38.51)
```

Name	Value
Global Variables	
Class Variables	
f	"(nat -> nat)"
self	"SelectionSort{#2, test:=undefined, data:=und
x	"1"

```
            dcl mi : nat := min_index(data(i,...,len data)) + i - 1;
            temp1 := data(mi);
            data(mi) := data(i);
            data(i) := temp1;
            SelectionSorter(i+1)
           )

functions
    public f: nat -> nat
        f(x) ==
        (
            if x = 1
                then 1
            else
                x * f(x-1)
        );

end SelectionSort
```

```
data_init(seq of (int))
sort_ascending()
sort_descending()
MergeSort
    sort_ascending()
    MergeSorter(seq of (real))
    Merge(seq of (int), seq of (int))
SelectionSort
    min_index(seq1 of (nat))
    startSort()
    print()
    sort_ascending()
    SelectionSorter(nat)
```

```
=>x
"1"
=>7 < 5
"false"
=>5 < 7
"true"
=>
```

Figure 1.6: Debugging perspective

1.4.2.1 Debug View

The debug View is located in the upper left corner in the Debug Perspective - see figure 1.6. 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.

1.4.2.2 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 arrays and so forth.

1.4.2.3 Breakpoints View

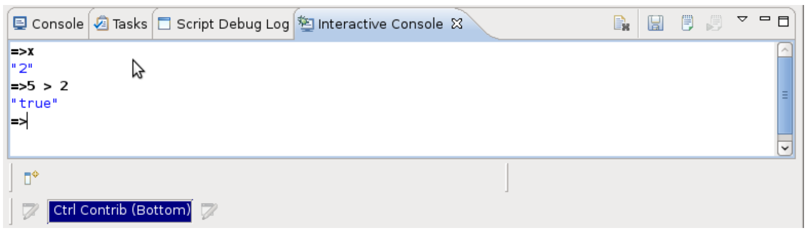
Breakpoints can be added both from the edit perspective and the debug perspective from the editor view. In the debug perspective however, there is a breakpoints view that shows all breakpoints. From the breakpoints view the user can easily navigate to the location of a given breakpoint, disable, delete or set the hit count or a break condition. In figure 1.6 the Breakpoints View is hidden behind the Variables View in the upper right hand corner in a tabbed notebook. Section 1.4.2.6 explains how to use conditional breakpoints.

1.4.2.4 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.

1.4.2.5 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 1.7 shows the interactive console.



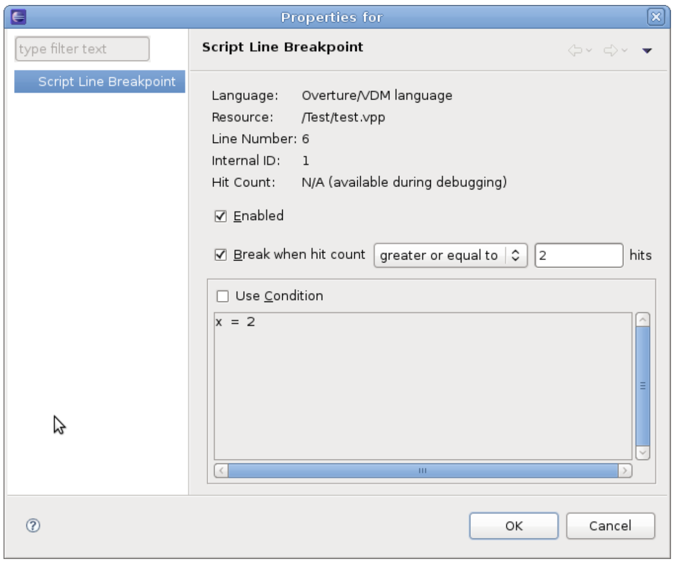
```
=>x
"2"
=>5 > 2
"true"
=>
```

Figure 1.7: The interactive console

1.4.2.6 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 1.8. It is possible to choose between two different conditional breakpoints, a hit count condition and one based on an expression defined by the user.



Script Line Breakpoint

Language: Overture/VDM language
Resource: /Test/test.vpp
Line Number: 6
Internal ID: 1
Hit Count: N/A (available during debugging)

☑ Enabled

☑ Break when hit count greater or equal to 2 hits

☐ Use Condition

```
x = 2
```

Figure 1.8: Conditional breakpoint options

1.5 Combinatorial Testing

A specific manual for this topic exist, please refer to [**?**].