Overture – Open-source Tools for Formal Modelling TR-2010-04
May 2014

Overture Development Documentation

by

Kenneth Lausdahl
Augusto Ribeiro

Contents

0.1 Document History

Revision	Notes	Date
0.1	Launcher and Editor sections added	02.2010

Chapter 1

Introduction

This document describes the way Overture is implemented in Eclipse.

Chapter 2

Logistics

In this chapter it is described what is the development environment of Overture and its supporting structure.

2.1 Overture Source Forge

2.2 Overture SVN

2.3 Maven

Maven[1] is a software tool for Java project management and build automation. It is similar in functionality to the Apache Ant tool, but is based on different concepts. Maven uses a construct known as a Project Object Model (POM) to describe the software project being built, its dependencies on other external modules and components, and the build order. It comes with pre-defined targets for performing certain well defined tasks such as compilation of code and its packaging [**?**].

2.3.1 Maven commands

2.4 Overture Development Environment

[1] `http://maven.apache.org/`

Chapter 3

Implementation

In this chapter is explained how different parts of Overture were implemented as well as important information on those different parts concerning extension, mantainence, tips, etc.

3.1 Editor

The idea behind the development of the VDM Eclipse editor was to define first an abstract editor that contains all the common functionality to the 3 VDM dialects and then let each of the dialects define its own concrete editor. In this way, it is easy to make extensions or even complete change some parts by extending the abstract and implementing/overriding methods. The implementation of parts of the editor was based on [**?**].

The eclipse text editor architecture is conceptually divided in two parts: the document and the viewer. The document contains the actual content of the editor while the viewer is responsible for the way to show this content.

3.1.1 Package editor.core

This is where the core definition of the VDM editor is made. First, it is defined a new abstract editor based on the provided Eclipse TextEditor.

```
public abstract class VdmEditor extends TextEditor {

        public VdmEditor()
        {
                super();
                setDocumentProvider(new VdmDocumentProvider());
```

```
        }
        ...
}
```

Listing 3.1: Declaring VdmEditor

The `org.eclipse.ui.editors.text.TextEditor` class ties a Document with a Viewer and adds Eclipse-specific functionality. For this we need to define a *DocumentProvider* and a Viewer. We defined our own *VdmDocumentProvider* of which we will talk later. In this piece of code we have set up the *DocumentProvider* but we are missing the setup of the *SourceViewer* which is made by overriding the method `createSourceViewer`.

```
@Override
protected ISourceViewer createSourceViewer(Composite parent,
        IVerticalRuler ruler, int styles) {

        ISourceViewer viewer = new VdmSourceViewer(parent,ruler,
                getOverviewRuler(), isOverviewRulerVisible(), styles,
                    this);

        return viewer;
}
```

Listing 3.2: VdmEditor: SourceViewer setup

We have also defined our own *VdmSourceViewer* which basically extends the Eclipse *SourceViewer* but this was made so that it can be easily changed in the future.

The other overridden method in this class is `initializeEditor`. This is method is where we set up the Viewer configuration.

```
@Override
protected void initializeEditor() {
        super.initializeEditor();
        setSourceViewerConfiguration(getVdmSourceViewerConfiguration()
            );
}

protected abstract VdmSourceViewerConfiguration
        getVdmSourceViewerConfiguration();
```

Listing 3.3: VdmEditor: `initializeEditor`

The operation that gets the *SourceViewer* configuration is made abstract because it will vary from dialect to dialect so each specific dialect edito plugin has to extend our *VdmEditor* class and provide an implementation for this method.

The *VdmSourceViewerConfiguration* class is one of the most important when defining the editor. This class is responsible for activating most of the editor functionality.

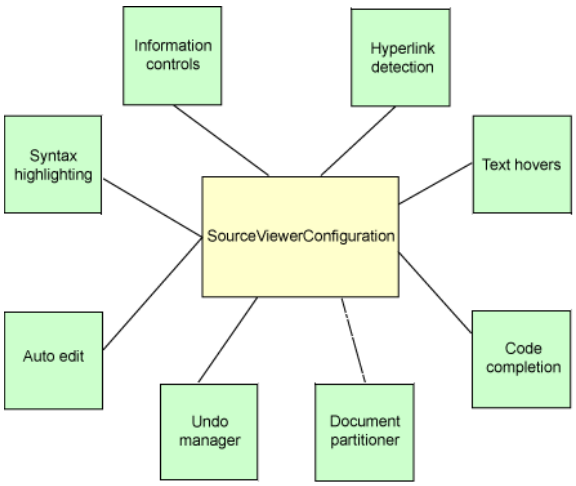


Figure 3.1: SourceViewerConfiguration needs to setup the different functionality

Figure **??** shows some of the editor functionality that needs to be configured in the *VdmSourceViewerConfiguration*. The configuration is changed by overriding the existing methods.

The *VdmDocumentProvider* bridges the file on the disk with its representation as a document in memory. We do this by extending the Eclipse class *FileDocumentProvider*.

```
public class VdmDocumentProvider extends FileDocumentProvider {
        ...
}
```

Listing 3.4: Snippets of `VdmDocumentProvider`

We then override two methods:

```
@Override
protected IDocument createDocument(Object element) throws
    CoreException {
```

```
  IDocument document = super.createDocument(element);

  if(document instanceof IDocumentExtension3)
  {
    IDocumentExtension3 extension3 = (IDocumentExtension3) document;
    IDocumentPartitioner partitioner =
      new VdmDocumentPartitioner(
        VdmUIPlugin.getDefault().getPartitionScanner(),
        VdmPartitionScanner.PARTITION_TYPES);
    extension3.setDocumentPartitioner(VdmUIPlugin.VDM_PARTITIONING,
      partitioner);
    partitioner.connect(document);
  }

  return document;
}
```

Listing 3.5: Snippets of `VdmDocumentProvider`

What we do here is creating a new document and connecting it to the partition scanner (the partition scanner is explained more in detail in the section **??**).

3.1.2 Package editor.partitioning

Each document is divided into one or more non-overlapping partitions. Many of the text-framework features can be configured to operate differently for each partition. Thus, an editor can have different syntax highlighting, formatting, or Content Assist for each partition [1] .

3.1.3 Package editor.autoedit

3.1.4 Package editor.syntax

3.2 Navigator

TBD

[1] `http://wiki.eclipse.org/FAQ_What_is_a_document_partition%3F`

3.3 Outline

TBD

3.4 Perspective

TBD

3.5 Launcher

The launcher is the part of the eclipse framework responsible to ”launch” the code to run, in our case a model to be interpreted. Associated with a launch is a launch configuration that contains information used to launch the code. The inspiration for the Overture implementation can be found here[2] [**?**]. Also connected with the launcher is its UI part which is the window that pops up when a defining a new launch configuration.

3.5.1 The players

launchConfigurationType: it is an extension point where new launch configurations can be declared. A launch configuration describes a way to launch a model;

launchConfigurationDelegate: it is a delegate associated with a launchConfigurationType. The delegate is in charge of, using the configuration set for launch, starting up the interpreter process. The configuration contains which is the launch mode (i.e. ”run”, ”debug”) among other settings;

launchConfigurationTypeImages: it is an extension point to select an image associated with a launch configuration type;

launchConfigurationTabGroups: it is an extension point that defines a tab group associated with a certain configurationType. The tab group is a group of tabs presented when creating a new launch configuration which contain the enables the user to graphically set the settings to be used in the launch (configuring it);

launchShortcuts: it is an extension point that enables the definition of shortcuts to launch models without configuring the launch, i.e. without bringing up the launch configuration tabs.

[2] `http://www.eclipse.org/articles/Article-Launch-Framework/launch.html`

launchGroups: didnt get there yet... not sure if it will be needed.

3.5.2 launchConfigurationType

To define a new launchConfigurationType this is the extension point.

```
<extension
        point="org.eclipse.debug.core.launchConfigurationTypes">
     <launchConfigurationType
           delegate="org.overture.ide.debug.core.launch.
              VdmLaunchConfiguration"
           id="org.overture.ide.debug.launchConfigurationType"
           modes="run, debug"
           name="Overture Launch Configuration Type"
           public="true">
     </launchConfigurationType>
  </extension>
```

Listing 3.6: lauchConfigurationTypes extension point

Important fields here are the

delegate: a class that implements *ILaunchConfigurationDelegate*. The *VdmLauchConfigurationDelegate* is located in the package `org.overture.debug.core.launching` and it contains the code that checks the given configuration and that launches the interpreter thread;

delegateName: humman readable name for the delegate;

id: a suitable ”id”;

modes: the modes which the launcher support, in our case: ”run” without debug and debug mode;

name: the human readable name for the configuration;

public: setting this attribute to true enables the launch configuration to be presented in the UI;

sourceLocatorId: we will come back to this attribute later in the debug section;

sourcePathComputerId: same as the one above.

There is only method to be written with the launch configuration, in the *VdmLauchConfigurationDelegate* and it is the method `launch`.

```
public void launch(ILaunchConfiguration configuration, String mode,
    ILaunch launch, IProgressMonitor monitor) throws CoreException {
         ...
         }
```

Listing 3.7: Snippets of VdmLauchConfigurationDelegate

The arguments are:

configuration: the configuration that the user selects in the UI is passed to the launch delegate through this variable;

mode: the mode in which the interpreter is supposed to run;

launch: created process and debug targets are added to this variable;

monitor: UI progress monitor of the launch;

In the `launch` method, the first thing to be done is to check if the actual configuration is correct. After these checks, the interpreter process is launched, if the debug mode is selected then the process should be attached to the debug target.

```
public void launch(ILaunchConfiguration configuration, String mode,
        ILaunch launch, IProgressMonitor monitor) throws CoreException
            {

        // initial checks
        ...
        // building process string
        String commandLine = "java.exe -cp vdmj-2.0.0.jar" + ...;
        ...
        if (mode.equals(ILaunchManager.DEBUG_MODE)) {
                // start intepreter in debug mode
                commandLine = commandLine + "debug options";
                // start the debugger
                Process process = Runtime.getRuntime().exec(
                    commandLine);
                // attach process to debug target
                IProcess p = DebugPlugin.newProcess(launch, process,
                    path);
                IDebugTarget target = new VdmDebugTarget(launch,p,s);
```

```
                launch.addDebugTarget(target);
                }
        }
```

Listing 3.8: Snippets of VdmLauchConfigurationDelegate

This is the simplified version of the launch delegate, in practise, because the Vdm interpreter tries to connect via a socket to the IDE when executed in debug mode, we have to start a socket server before launching the process.

Tip 1: the launch configuration (`configuration`) is still accessible while the debugger is running via the DebugPlugin. It is possible to set attributes that can be used later in the debug.

```
        ...
        ILaunchConfigurationWorkingCopy workConfiguration =
                configuration.getWorkingCopy();
        workConfiguration.setAttribute("attribute","value");
        configuration = workConfiguration.doSave();
        ...
```

Listing 3.9: Changing the configuration

Tip 2: use the constants in ILaunchManager to compare with the launch mode

```
        ...
        if (mode.equals(ILaunchManager.DEBUG_MODE)) {
        ...
        }
        ...
```

Listing 3.10: Comparing the launch mode

3.5.3 launchConfigurationTypeImages

The `launchConfigurationTypeImages` extension point is a way to assign a icon to a launch configuration.

```
<extension
         point="org.eclipse.debug.ui.launchConfigurationTypeImages">
      <launchConfigurationTypeImage
```

```
            configTypeID="org.overture.ide.debug.
                launchConfigurationType"
            icon="icons/cview16/overture_nav.gif"
            id="org.overture.ide.debug.launchConfigurationTypeImage">
        </launchConfigurationTypeImage>
</extension>
```

Listing 3.11: launchConfigurationTypeImages extension point

Fields:

configTypeID: the id of the `launchConfigurationType` the image is used for;

icon: the icon to be used;

id: an id for this `launchConfigurationTypeImages`.

There is not much more to say about this extension point. After defining a `launchConfigurationType` and a `launchConfigurationTypeImages` for it, the UI when the *Debug Configurations...* button is pressed shows like in figure **??**.

3.5.4 Laucher UI part

After defining the 2 extension points above, we have a launch type and a launch icon, but normally we would like to receive some user input (configuration) of the model to launch, such as: which is the entry point method of the model, if the user wants test coverage and so on. These options should be presented in the *Debug Configuration* (figure **??**) window when pressing our type of configuration. This is made by declaring a `launchConfigurationTabGroup`. Fields:

Extension point 1 launchConfigurationTabGroup extension point

```
<extension
         point="org.eclipse.debug.ui.launchConfigurationTabGroups">
      <launchConfigurationTabGroup
            class="org.overture.debug.ui.tabs.VdmLaunchConfigurationTabGroup"
            description="Vdm Launch Config"
            id="org.overture.debug.vdmLaunchConfigurationTabGroup"
            type="org.overture.debug.vdmLaunchConfigurationType">
      </launchConfigurationTabGroup>
</extension>
```

class: the class implementing the tab group;

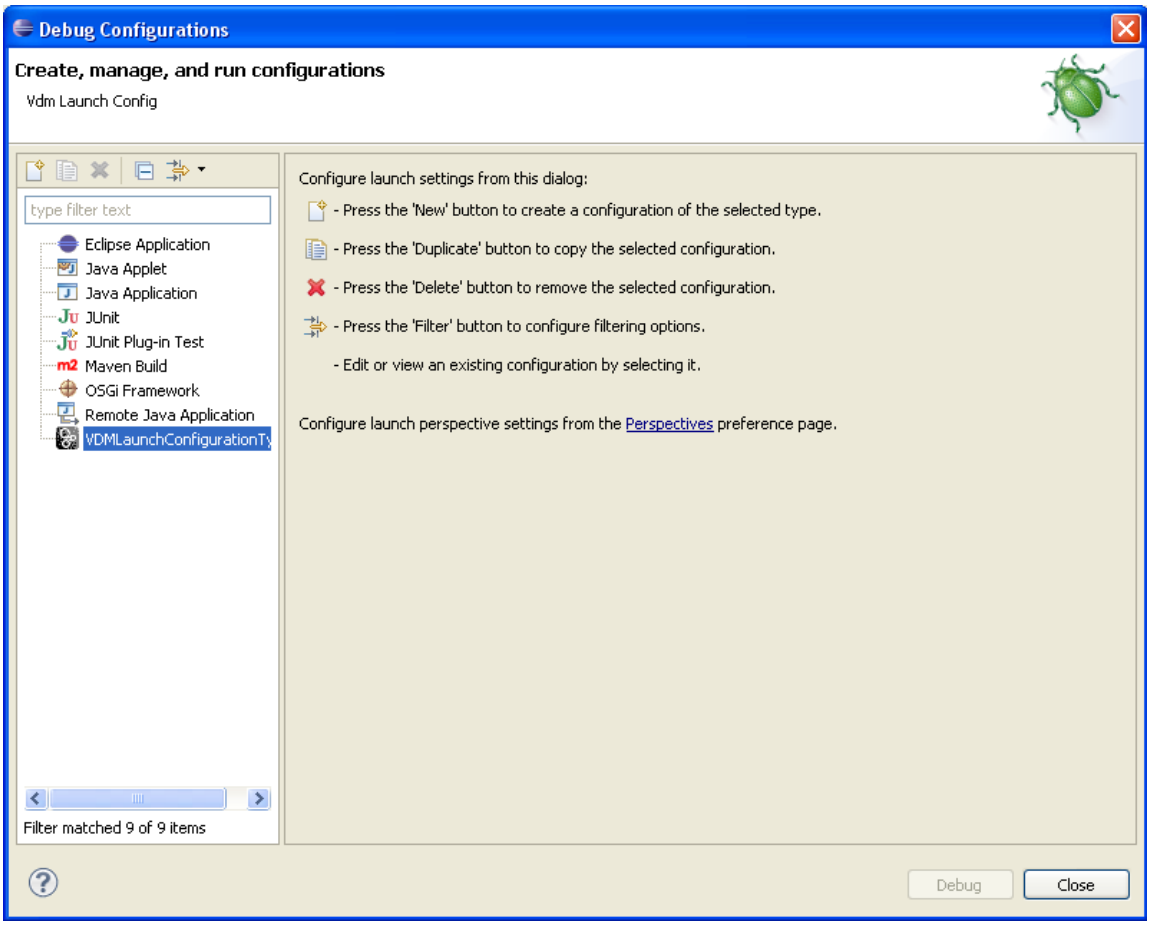


Figure 3.2: VDM Launch Configuration

description: a human readable description;

id: a id for the tab group;

type: the id of the `launchConfigurationType` we want to define the tabs for.

Here there is also the task of defining one new class *VdmLaunchConfigurationTabGroup* in which only one method needs to be defined if it extends the *AbstractLaunchConfigurationTabGroup* class that eclipse provides.

```
        ILaunchConfigurationTab[] tabs = new ILaunchConfigurationTab
            []{
```

```
                                new VdmppMainLaunchConfigurationTab(
                                    mode),
                                new VdmInterpreterTab(),
                                new SourceLookupTab(),
                                new CommonTab()
                        };
                        setTabs(tabs);
```

Listing 3.12: createTabs method in *VdmLaunchConfigurationTabGroup*

Then the only task is to define the tabs that we would like to provide to the user. These user defined tabs should either extend the *AbstractLaunchConfigurationTab* or implement *ILaunchConfigurationTab*. Eclipse guidelines also advise to include always the *CommonTab* as one of the tabs and also the *SourceLookupTab* if we use source lookup, which will omit by now but will introduced on the debug section. The tabs defined in this class appear in the *Debug Configuration* window as shown in figure **??**.

Launch configuration shortcuts

TBD.

3.6 Debugger

This section explains how the Overture debugger is constructed. It was build inspired in this article[3] [**?**].

3.6.1 The players

The debug model defines several entities which should be present in the debug. To incorporate the eclipse debug model we need to implement some classes which are described below:

IDebugTarget: is the interface implemented by *VdmDebugTraget* and it represents the target in which the model is running, a kind of virtual machine, which can be started and stopped. The class represents the debug environment and it contains the debug threads;

IThread: implemented by *VdmThreads* represents a thread running the model. A thread contains a stackframe;

[3] `http://www.eclipse.org/articles/Article-Debugger/how-to.html`

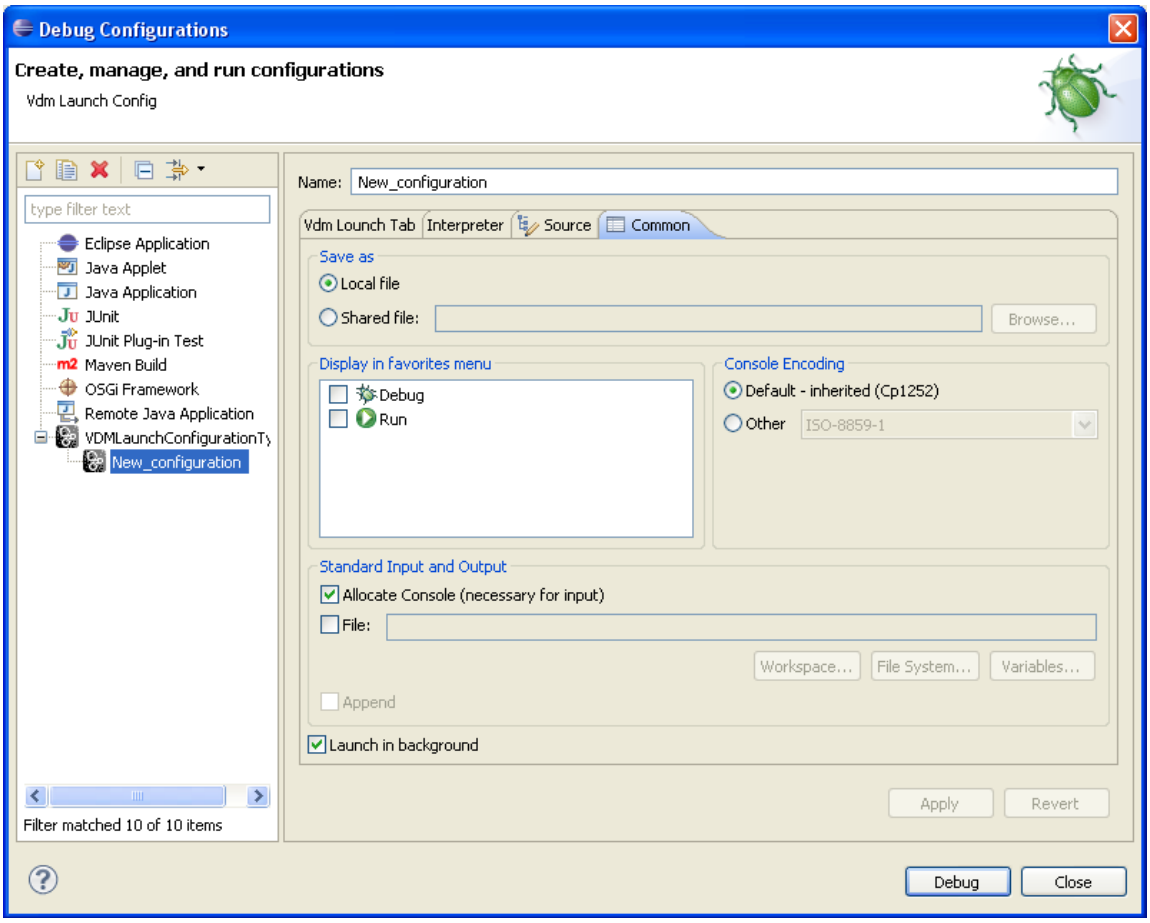


Figure 3.3: VDM Launch Configuration Tabs

IStackFrame: implemented by *VdmStackFrame*, it contains information about the call stack and the context variables;

IVariable: implemented by *VdmVariable*, it contains variable values;

IValue: implemented by *VdmValue*, it contains values or variables. This item and the on above form a treelike structure to represent more complex variables;

IDebugElement: all of the classes above inherit from this interface. It contains common functions shared by all of them;

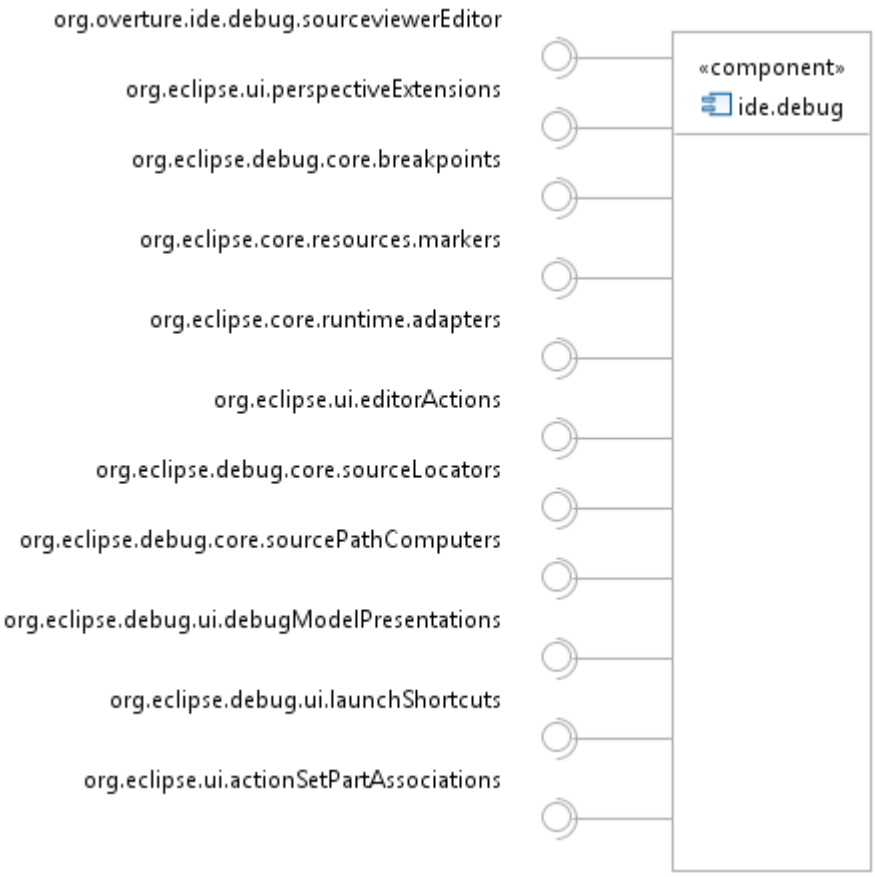

org.overture.ide.debug.sourceviewerEditor
org.eclipse.ui.perspectiveExtensions
org.eclipse.debug.core.breakpoints
org.eclipse.core.resources.markers
org.eclipse.core.runtime.adapters
org.eclipse.ui.editorActions
org.eclipse.debug.core.sourceLocators
org.eclipse.debug.core.sourcePathComputers
org.eclipse.debug.ui.debugModelPresentations
org.eclipse.debug.ui.launchShortcuts
org.eclipse.ui.actionSetPartAssociations
«component»
ide.debug

Figure 3.4:

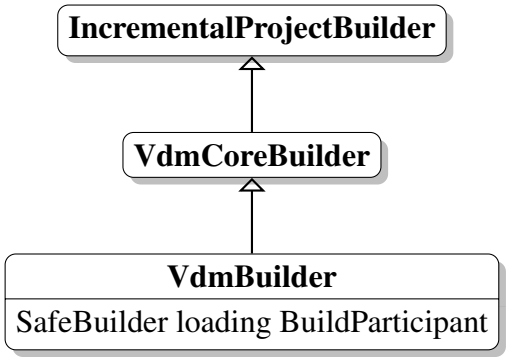
IBreakpoint: implented by *VdmBreakpoint* contains information on hwo to break a running model to send to the debugger.

A visual representation of this relation is shown on figure **??**.

3.6.2 VdmDebugTarget

This is the class the implements the ”virtual machine” where the model to be debugged is running.

3.7 Builder




```
<!-- Basic builder for VDM -->
  <extension
        id="org.overture.ide.core.builder.VdmBuilder"
        point="org.eclipse.core.resources.builders">
     <builder
           callOnEmptyDelta="false"
           hasNature="false"
           isConfigurable="true">
        <run
              class="org.overture.ide.core.builder.VdmBuilder">
        </run>
     </builder>
  </extension>
```

Listing 3.13: createTabs method in *VdmLaunchConfigurationTabGroup*

```
<!-- Extension point for builders of VDM -->
 <extension-point id="org.overture.ide.core.buildParticipant" name="
    Builder" schema="schema/org.overture.ide.core.builder.exsd"/>
```

Listing 3.14: createTabs method in *VdmLaunchConfigurationTabGroup*

```
<schema targetNamespace="org.overture.ide.core" xmlns="http://www.w3.
   org/2001/XMLSchema">
<annotation>
      <appinfo>
         <meta.schema plugin="org.overture.ide.core" id="org.overture.
            ide.core.buildParticipant" name="Builder"/>
      </appinfo>
   </annotation>

   <element name="extension">
      <complexType>
         <choice minOccurs="1" maxOccurs="unbounded">
            <element ref="builder"/>
         </choice>
         <attribute name="point" type="string" use="required">
            <annotation>
            </annotation>
         </attribute>
         <attribute name="id" type="string">
            <annotation>
            </annotation>
         </attribute>
         <attribute name="name" type="string">
            <annotation>
               <appinfo>
                  <meta.attribute translatable="true"/>
               </appinfo>
            </annotation>
         </attribute>
      </complexType>
   </element>

   <element name="builder">
      <complexType>
         <attribute name="class" type="string">
            <annotation>
               <appinfo>
```

```
                    <meta.attribute kind="java" basedOn="org.overture.
                        ide.core.builder.BuildParcitipant:"/>
                </appinfo>
            </annotation>
        </attribute>
    </complexType>
  </element>
</schema>
```

Listing 3.15: createTabs method in *VdmLaunchConfigurationTabGroup*

3.8 Parser

```
<!-- Extension point for parsers of VDM -->
<extension-point id="org.overture.ide.core.parsePaticipant" name="
    Parser" schema="schema/org.overture.ide.core.parsePaticipant.
    exsd"/>
```

Listing 3.16: createTabs method in *VdmLaunchConfigurationTabGroup*

3.9 Parser

TBD

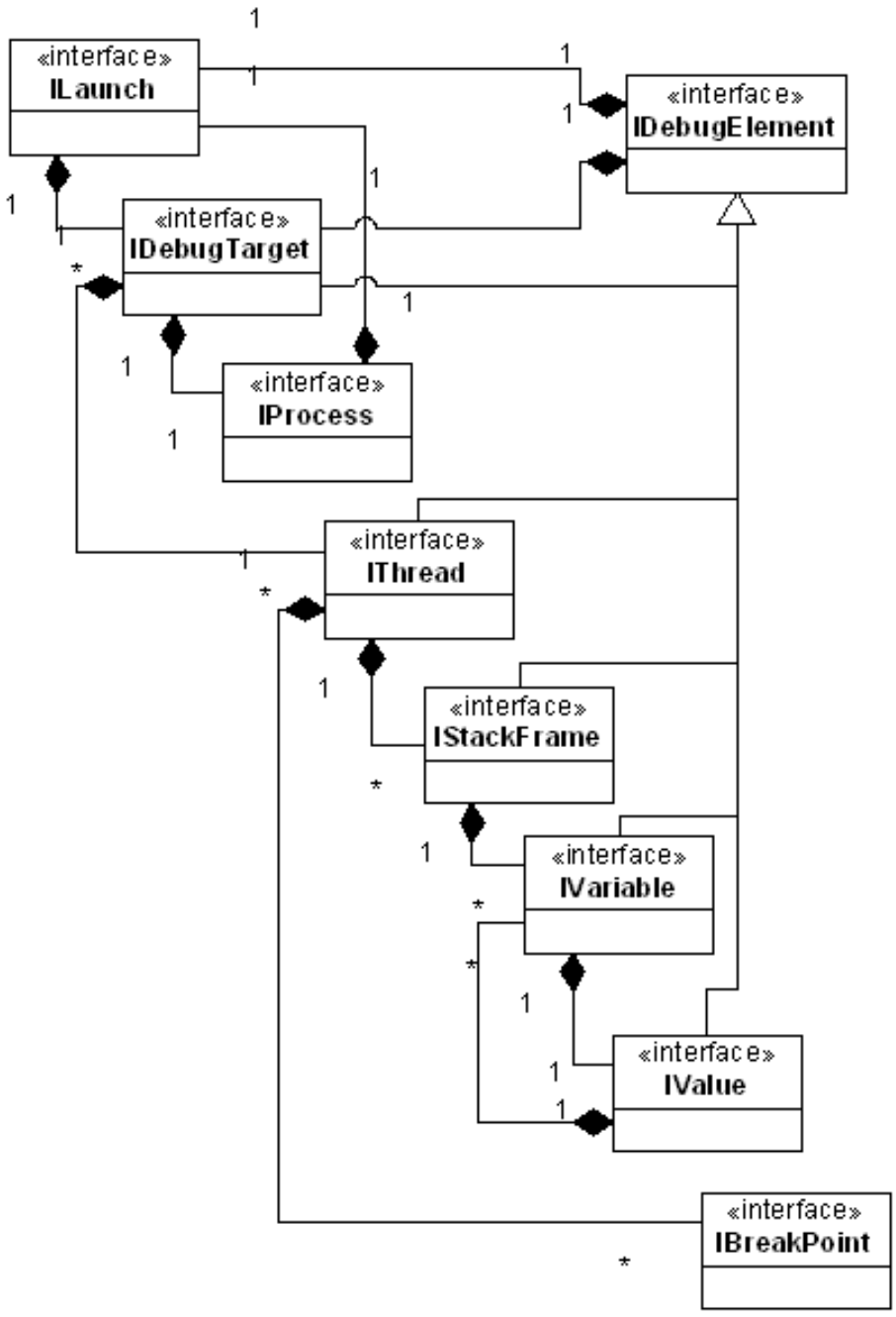


Figure 3.5: Eclipse debug model