Modelica Development Tooling for Eclipse

Elmir Jagudin
Andreas Remar

Contents

1 Introduction

The creation of software is a complex and error prone task. To help the programmers, tools have been developed that assists with the making of software. A programmer needs many tools to develop software in an efficient way, some of these are editors, compilers, and debuggers. To utilize all these tools in a nice way, and to get a better workflow, the tools are integrated into a so called Integrated Development Environment (IDE).

An IDE is a collection of development tools glued into one big program, so that they can be reached easily. For example, if the programmer would like to stop editing and start compiling a program, she could just press the Compile button instead of exiting the editor and giving the compile command. This compilation could even be automatic (e.g. when the user saves a file) and immediately inform the programmer when an error has been typed. This will hopefully save time and allow the programmer to focus on the problem she's trying to solve.

Two of the more popular IDE's today are Visual Studio and Eclipse. Visual Studio is Microsoft Software's IDE, and is an IDE for C++, Visual Basic, C#, and J#. As it is proprietary software, it's not of much interest to this thesis. Eclipse is, at least from the beginning, an IBM project that later got released to the public as free software[3]. As of now, the Eclipse Foundation manages the Eclipse project. Eclipse is an "IDE for everything and nothing in particular". However, it is shipped by default with a set of plugins that has very good support for developing Java projects.

As Eclipse is a very good platform for creating even better IDE's, we developed the Modelica Development Tooling with and for Eclipse. As Eclipse has a plugin architecture where different parts are easily replacable, it's possible to incrementally build a development environment.

MDT (the Modelica Development Tooling) is a collection of plugins for Eclipse that provides an environment for working with Modelica projects. MDT integrates with the OpenModelica Compiler to provide support for various features, for example package and class browsing and code completion. These features will hopefully make it easier for the model designer to create Modelica models.

1.1 Intended audience

The following report on the Modelica Development Tooling assumes that the reader have some understanding of how plugins for Eclipse are implemented. Familiarity with the Modelica and Java languages is recommended. Some CORBA terminology is used towards the end of the report.

Contributing to Eclipse[16] is a good introduction on writing plugins for Eclipse. A bit bulky but thorough work on the Modelica language is Peter Fritzson's book[13]. Chapter two "A Quick Tour Of Modelica" is sufficient reading for understanding this text. As far as the authors know, no good books exists on CORBA.

1.2 Thesis contributions

This thesis gives an overview of the Modelica Development Tooling for Eclipse that was developed at PELAB. As there were no real scientific breakthroughs while doing this thesis, no scientific contributions are made available.

1.3 Thesis outline

The following is a short outline of the thesis:

Chapter 2 starts off with a short background for Modelica, Eclipse and the Open Modelica Compiler.

Chapter 3 has a quite detailed description of the architecture of the Modelica Development Tooling plugins.

Chapter 4 has a discussion about the testing framework and testing tools that were used when developing MDT.

Chapter 5 contains some suggestions on future work.

Chapter 6 has discussions regarding the Open Modelica Compiler, the testing of GUI code, and the Modelica Compiler interface.

Chapter 7 concludes this thesis by detailing what we've accomplished and what we deliver.

2 Background

To get a grasp of what this thesis is about we'll first have a brief introduction to Modelica and Eclipse. We will also briefly discuss the OpenModelica Compiler, as it is an important component of the Modelica Development Tooling.

2.1 Modelica

Modelica[6][13] is an object-oriented programming language for modeling systems that can be described using mathematical equations. As systems in real life can be described by equations, they can be simulated using Modelica. Some of the most important features of Modelica are:

- Models in Modelica can be described using equations instead of algorithms. This means that the flow of the data is not specified, which leads to better model reuse.
- Multidomain modeling, meaning that you can mix components from different domains such as electrical, mechanical, and biological. This brings great flexibility as you can specifiy large systems that contains parts from many different areas of physics.
- Modelica has a general class concept, which further simplifies the reuse of code in models.

See the tutorial by Modelica Association[10] for an introduction to Modelica model development.

2.2 Eclipse

Eclipse[2] is an open source framework for creating extensible integrated development environments (IDEs). The integration simplifies development and avoids disturbing the "flow"[12] that a programmer can attain when programming.

By itself, Eclipse doesn't provide alot of end-user functionality. The greatness of Eclipse is based on the plugins that are plugged in. The smallest unit of the Eclipse platform is the plugin.

2.2.1 Eclipse Platform Architecture

At the core of Eclipse is the Eclipse Platform Runtime. The Runtime in itself mostly provides the loading of external plugins. The Java Development

Tooling is for example a collection of plugins that are loaded into Eclipse when they are requested. That Eclipse is in itself written in Java and comes with the Java Development Tooling as default often leads newcomers to believe that Eclipse is a Java IDE with plugin capabilites. It is in fact the other way around, with Eclipse being just a base for plugins, and the Java Development Tooling plugging into this base. See figure 1 on page 8.

To extend Eclipse, a set of new plugins must be created. A plugin is created by extending a certain extension point in Eclipse. There are several predefined extension points in Eclipse, and plugins can provide their own extension points. This means that you can plug in plugins into other plugins.

An extension point can have several plugins attached, and what plugin that will be used is determined by a property file. For example, the Modelica Editor is loaded at the same time as the Java Editor is loaded. When a user opens a Java file, the Java Editor will be used, based on a property in the Java Editor extension. (In this case, it's the file ending that determines what editor that should be used.)

As the number of plugins in Eclipse can be very big, a plugin is not actually loaded into memory before its contribution is directly requested by the user. This design assures us that the memory impact will be as low as possible while running Eclipse.

A user-friendly aspect of Eclipse is the Eclipse Update Manager which allows you to install new plugins just by pointing Eclipse to a certain website. This website is provided by the developers of the plugin that you may wish to install. An update site is for example provided by the MDT Development Team for easy installation of the latest version of MDT.

2.3 OpenModelica Compiler

The OpenModelica Compiler (OMC) is being developed at PELAB. It is a part of an effort to produce a complete Modelica environment[7] for creating and simulating Modelica models.

OMC keeps a representation of every model in memory so that they can be queried interactively by the user. When defining a new model, it is sent to OMC via the provided interactive API, see section 3.2 on page 13. This interactive API is then used to provide MDT with information about Modelica models and packages. This information is for example utilized in the MDT interface for providing a tree view of packages and models.

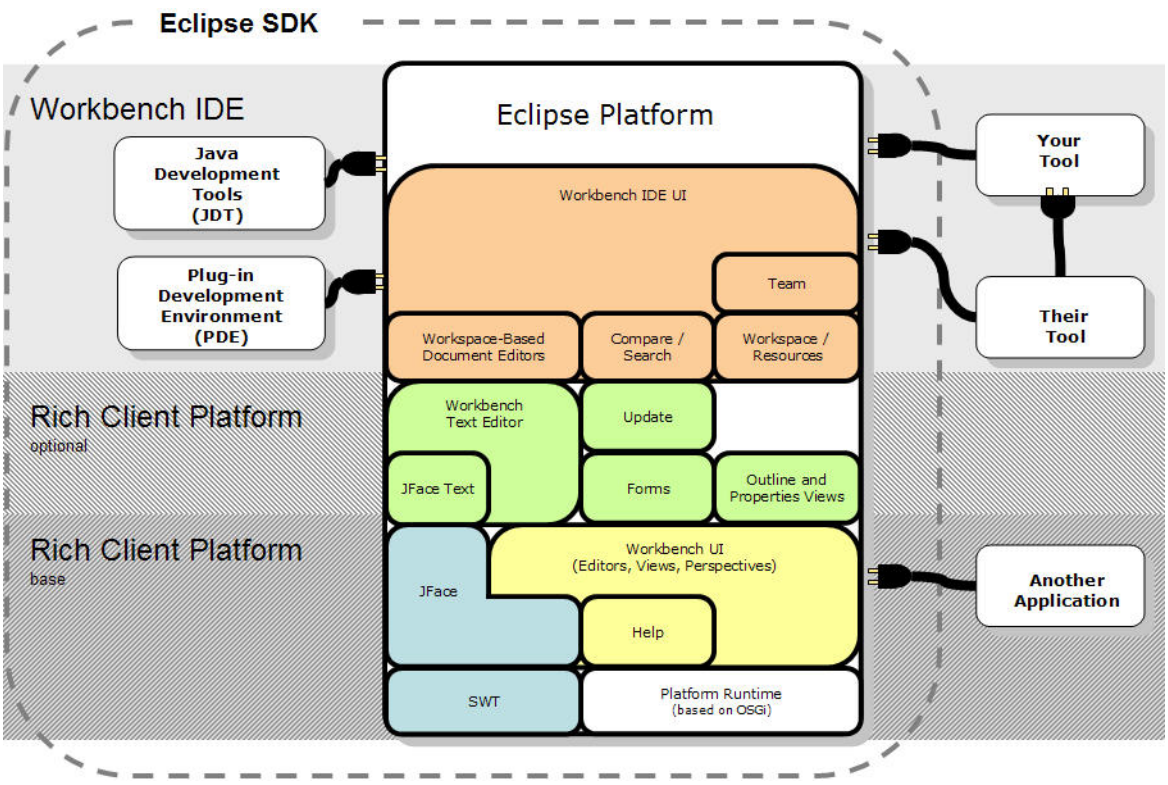

Figure 1: Eclipse Platform Architecture

3 Architecture

The Modelica Development Tooling package is composed out of three separate Eclipse plugins. These three plugins are *org.modelica.mdt.core*, *org.modelica.mdt.ui* and *org.modelica.mdt.omc*. These plugins contributes respectively core Modelica functionality, user interface and OpenModelica Compiler access services. Together these plugins adds Modelica-specific functionality to the Eclipse IDE and creates an environment for working on Modelica projects.

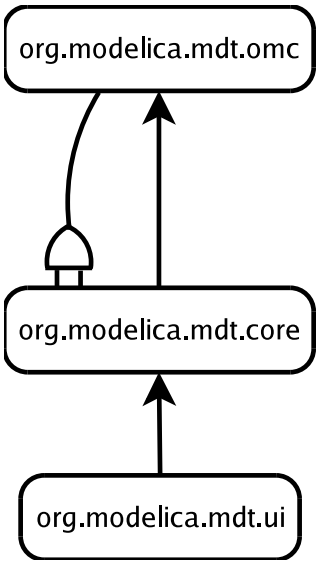

Figure 2: MDT Plugins Architecture

The omc plugin plugs in into the core plugin to provide compiler services. The ui plugin uses the services provided by the core plugin. To fullfill some requests the core plugin must employ services provided by the omc plugin.

3.1 org.modelica.mdt.core

The core plugin provides main functionality for MDT. It allows browsing Modelica components hierarchies and querying a mapping to the source code. It also provides a mechanism to track changes to components.

3.1.1 Component access layer

The classes and interfaces that provides access to the Modelica components are defined in the *org.modelica.mdt.core* package. All client plugins should only use the API defined in that package to access Modelica components.

The components contained in each project are made accessible by wrapping the *IProject* object with an instance of the *IModelicaProject* class. To obtain a wrapped versions of all projects in the workspace the method *IModelicaRoot.getProjects()* should be used. To obtain an instance of the *IModelicaRoot* interface the static method *ModelicaCore.getModelicaRoot()* is available.

3.1.2 Modelica projects

IModelicaProject provides access to the wrapped version of its root folder via the *getRootFolder()* method. The method returns an instance of the *IModelicaFolder* interface. All Modelica resources, and other types of resources, are contained in the root folder of their respective project.

3.1.3 Folders

IModelicaFolder is a wrapper for a regular folder represented by an instance of *IFolder*. A Modelica folder can contain, besides other folders and plain files, Modelica source code files and packages. Any file with the extension *mo* in the file name is treated as a Modelica source code file.

The Modelica language specification defines a standard way to map a Modelica package to a folder structure[14]. Any such folders are so called folder packages and are treated as packages by *IModelicaFolder*.

Modelica source files are represented by the *IModelicaFile* interface, Modelica packages are represented by implementations of the *IModelicaClass* folder. The reason that there is no special interface type for a Modelica package is due to the fact that Modelica packages are defined in the language as regular classes of the special restriction type *package*.

3.1.4 Files

A Modelica source code file contains hierarchies of Modelica classes. The list of references to top-level classes are provided by the method *IModelicaFile.getChildren()*.

3.1.5 Classes

The Modelica language defines 7 restriction types of classes. These restrictions are *model*, *function*, *record*, *connector*, *block*, *type* and *package*. The restriction type of a class defines a restriction on the structure of the class. There is also a special restriction type called *class* which means that there

are no restrictions on the contents of the class. See a Modelica language manual for more information on the restriction types.

Classes of all restriction types are represented by the *IModelicaClass* interface. The method *getRestrictionType()* can be used to query for the restriction type of the class. A class can contain a myriad of Modelica elements, however at this point only subclasses and components are made accessible. Contents of a class are accessible via the *getChildren()* method, which returns a list of top-level subclasses and components.

3.1.6 Components

A component in the Modelica language can be compared to a member variable in a conventional object oriented programming language. Modelica defines two levels of visibility of class components, *public* and *protected*. The visibility affects how the components can be accessed outside of the class definition. See [15] for more information on component visibility.

Components are represented by the *IModelicaComponent* interface. Currently only the visibily and name of a component can be accessed via the *getVisibility()* and *getElementName()* methods.

3.1.7 Mapping to the source code

All interfaces that represent Modelica elements are derived from the common grandparent *IModelicaElement*. This interface defines methods to query for common attributes of Modelica elements, for example the element's name via *getElementName()*.

IModelicaElement also defines methods that allow determining the source code location where the element is defined. *getResource()* returns the resource where the element is defined. This can either be a folder or a file based on the type of the Modelica element. If the element is defined outside of the workspace, for example a system library element, *getResource()* returns a *null* value. When such information is available, the path to the source code file can be obtained with the *getFilePath()* method.

For elements that are defined inside a file the method *getLocation()* returns the region of the file where the element is defined. It should be noted that currently, due to limitations in OMC, the region returned is the first line of the definition rather than the complete definition.

3.1.8 Modelica Standard Library

The Modelica specification defines a standard library of packages. To provide access to packages in the standard library, the method *getStandardLibrary()*

in the *IModelicaRoot* interface is defined. The method returns a list of top level packages in the standard library. The exact list of packages returned is determined by the current compiler plugin.

3.1.9 Tracking element changes

To allow tracking changes to Modelica elements, clients can register a listener. Such a listener must implement the *IModelicaElementChangeListener* interface. *IModelicaRoot.addModelicaElementChangeListener()* can be used to register a listener. Whenever clients wish to stop receiving notification on element changes, the *removeModelicaElementChangeListener()* method can be employed.

The method *elementsChanged()* on the listener will be invoked whenever changes to the Modelica elements are detected and a list of changes are passed along. Each change to an element is encoded as an instance of the *IModelicaElementChange* interface. Such an object contains information on the changes nature and the element that have been changed. The change nature is one of added, removed or modified. For project elements there are also changes of the type opened and closed defined. On change type added a parent element of the newly added element is accessible via the *getParent()* method.

It should be noted that unlike Eclipse resource deltas, an element change list is a flat structure. No hierarchical information is made available.

3.1.10 Compiler extension point

The core plugin defines an extension point *org.modelica.mdt.compiler*. This extension point is used by the core plugin to load the class that is used to access a Modelica compiler. Currently the core plugin only accepts a single plugin that extends the compiler point. If there is none or more than one extension of the compiler extension point the core plugin returns an error to the clients on any calls that require access to a Modelica compiler.

The extension point requires that the extender specifies a class that will provide interface to the Modelica compiler. The specified class must implement the *org.modelica.mdt.compiler.IModelicaCompiler* interface. The core plugin will create an instance of the specified class via its default constructor and invoke the methods as defined in the interface to access the compiler. See the source code documentation of the *IModelicaCompiler* interface and *org.modelica.mdt.compiler* extension point documentation for details on implementing a compiler plugin.

3.2 org.modelica.mdt.omc

The OMC plugin provides access to the OpenModelica Compiler (OMC) for the core plugin. It does that by implementing the *org.modelica.mdt.compiler* extension point. The class that extends the extension point is called *OMCProxy*. The plugin acts as a proxy and redirects all the requests to OMC and translates back the replies for the core plugin.

3.2.1 Communicating with OMC

OMCProxy is the main class of *org.modelica.mdt.omc*. This class takes care of starting, connecting to and communicating with OMC. To communicate with OMC, a CORBA interface is utilized. This interface has a single function that takes a String as argument and returns a String containing the OMC reply. If OMC can't be found when trying to contact it, it will have to be started.

3.2.2 Starting OMC

If an OMC session cannot be found when communication with OMC is needed, a new session will be started. This is handled by the *startServer()* method in *OMCProxy*. This method uses the OPENMODELICAHOME environment variable to determine where the OMC executable can be found.

3.2.3 OMC interactive API

The interactive OMC API is entirely textual. This means that commands must be formulated as strings, and returned strings must be parsed to be able to get the actual returned information. The next subsection describes a parser for parsing returned strings.

The API is called an interactive API as it can be used interactively by a program (or a user) to query OMC for information about the contents of its database of stored models.

By using the API you can load models into OMC and get information about previously loaded models. This information is used by MDT for providing a range of features, for example to provide a tree view of packages and models, code completion when typing in code and finding and reporting errors found in models.

3.2.4 OMC communication parser

The OMC communication parser is quite simple. The returned string from OMC either contains a list of objects, or a list of errors. These lists look a bit different, but are quite easy to parse.

The list of objects is the most difficult to parse as there can be lists within a list. This recursivity makes it hard to just split the string on some given character or character sequence. Instead the parser tries to match up parentheses and sending each substring recursively to the parser. The parsed list is represented as a standard Java Vector. The method for parsing lists is called *parseList()* and resides in the *ModelicaParser* class found in the core plugin.

The list of errors are much easier to take apart as it's just a newline-separated list of errors in a specific format. Each error is easily parsed as long as it follows the standard error format in OMC. It turns out that many error messages from OMC don't follow any format, and will therefore not be parsed (and thereby generating an error). The error parser method is called *parseErrorString()* and can be found in the *OMCParser* class in the omc plugin.

3.2.5 Interfacing with the core plugin

To be able to get information to the core plugin about models, a few functions exist in *OMCProxy*. These functions are mapped relatively directly to OMC API function calls.

The method *getClassNames()* will return the names of classes and packages that are contained in a class or a package. It uses the OMC API function call with the same name.

The method *getRestrictionType()* will ask OMC about what kind of restrictions a class has. A restriction can, for example, be class, model, package, or function.

The method *loadFileInteractive()* takes a file as argument and tries to load that into OMC. This method will return a list of classes and packages in the file.

The method *getElementLocation()* will try to locate where in a file a class is defined. This is for example used when clicking a classname in the package browser that opens the editor with the correct file at the correct position.

The (predicate) method *isPackage()* simply asks OMC if a given classname is a package. This is a specialized version of the method *getRestrictionType()*.

The method *getElementsInfo()* gets information about the elements that

are contained in a class. In a class, there can be both elements and annotations, and this function returns both kinds as a long list. The types of elements that can be contained in a class are *classdef* (definition of a class), *extends* (what this class extends), *import* (what this class imports), and *component* (a component is kind of a member variable). For all these kinds of elements, the file and line number can be retreived. Alot more information is available from this API function call, but are not used in MDT. See the documentation contained in the OMC source tree for a longer explanation of this and other API function calls.

3.3 org.modelica.mdt.ui

The ui plugin implements the graphical interface for working with Modelica resources. The ui plugin uses the services provided by the core plugin. In a sense the ui plugin extends the core plugin by adding a user interface to the core Modelica services.

3.3.1 Modelica development ui

Most of the functionality provided by the ui plugin is grouped in the Modelica perspective, see figure 3.

The Modelica perspective contains (a) the Modelica projects browser, (b) the Modelica source code editor and (c) the problems view. The ui plugin also provides wizards to create new Modelica projects, classes and packages.

3.3.2 Modelica projects browser

The ui plugin contributes a Modelica projects view. The view is largely inspired by the Java package browser. The Modelica projects viewer allows the user to browse projects, folders, packages, source code files and classes. It also provides cognitive shortcuts to open the source code in the editor, via the double click mechanism, and wizards to create new elements, via the context menu.

The projects view displays a tree of projects in the workspace, basically the same way that the Java package view does. The tree presents the hierarchical structure provided by the core plugin graphically. By using the mapping to the source code it enables a fast way to open the underlying source code for reading and modifing in the Modelica text editor.

The code that implements the view is housed inside the *org.modelica.mdt.ui.view* package. The project views code structure is largely architectured after the

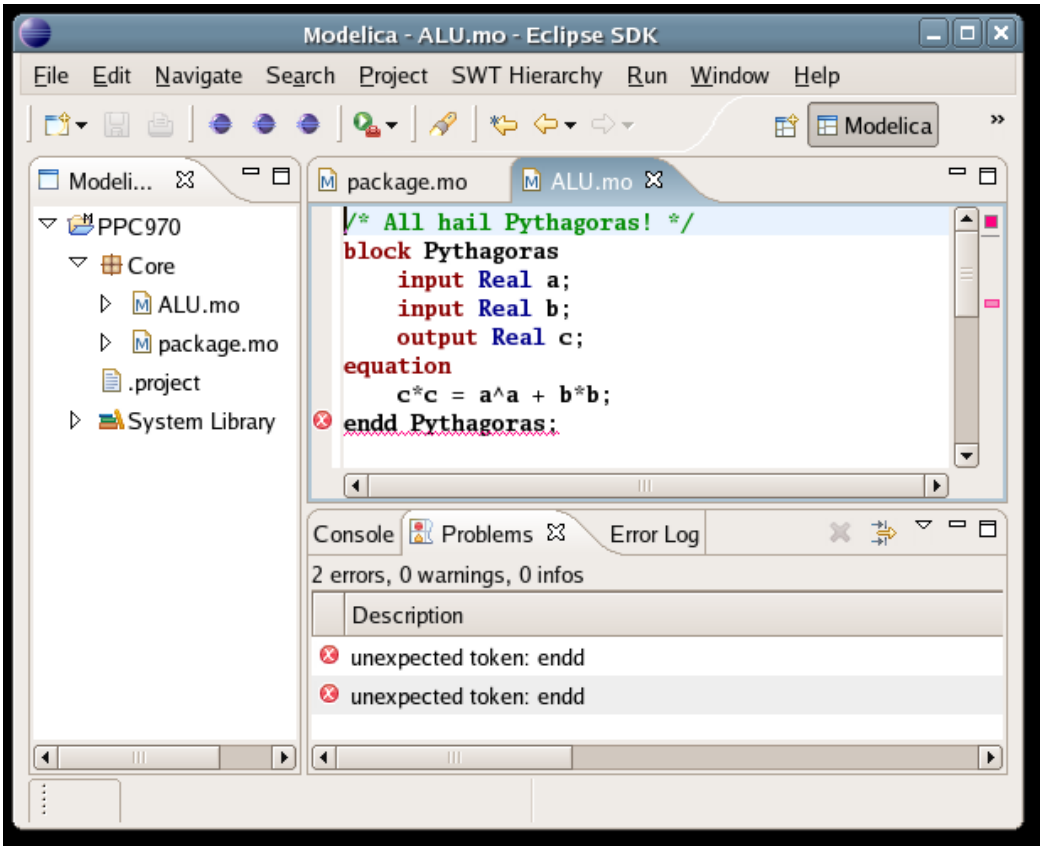

Figure 3: Modelica Perspective

resource navigator view code. The class *ProjectsView* is the class that implements the view part interface. It initializases the view and sets up the listeners and actions to animate the view.

The *ProjectsView* class sets up a tree viewer. It configures the tree viewer to use an instance of the *ModelicaElementContentProvider* class as a content provider, and the standard instance of *IModelicaRoot* as the input source. The labels and icons of the elements in the tree are provided by an instance of *WorkbenchLabelProvider* from the *org.eclipse.ui.model* package.

The *ModelicaElementContentProvider* provides contents by simply exposing the resource tree as presented by the core plugin via the *IModelicaRoot* interface. The *ModelicaElementContentProvider* also updates the tree viewer when the underlying Modelica elements changes. It does that by registering itself as a Modelica element change listener with *IModlicaRoot* in its constructor.

The *WorkbenchLabelProvider* provides labels and icons for elements in the

tree by trying to convert them to *IWorkbenchAdapter* via the Eclipse adapter mechanism, see [17] for more information. The ui plugin makes it possible to convert Modelica elements by installing an instance of *ModelicaElementAdapterFactory* as an adapter factory. This is done in the *Plugin.start()* method. When the *WorkbenchLabelProvider* tries to convert a Modelica element to *IWorkbenchAdapter*, an instance of *ModelicaElementAdapter* is returned by the adapter factory. *ModelicaElementAdapter* implements a mapping between Modelica elements and text labels and icons for graphical representation according to the definition of the *IWorkbenchAdapter* interface.

3.3.3 Opening elements in an editor

As noted earlier it is possible to open the source code of Modelica elements directly from the projects view by invoking an open action, usually double clicking on the element. This functionality is implemented by adding an anonymous open listener on the projects tree viewer. This listener invokes the method *handleOpen()*. The method retrieves the element that the open action was invoked upon and forwards it to the *openInEditor()* method in the *EditorUtility* class in the *org.modelica.mdt.ui.editor* package.

openInEditor() handles the details of opening an element in a correct editor. Non-Modelica elements are opened in their respective default editor. When a request is made to open a Modelica element, the method tries to determine the source file and region in the file where the element is defined. On success the file is opened and the region is sharklighted.

It should be noted that there is a separate double click listener registered on the Modelica projects tree viewer, an instance of the class *ProjectsViewDoubleClickAction*. However this listener only adds behaviour to the tree where some elements expand and collapse on double click. Do not confuse it with the open action listener!

3.3.4 Wizards

The wizards that the ui plugin contributes are implemented in the *org.modelica.mdt.ui.wizards* package. The wizards are made accessible to the user via Eclipse's standard wizards access points, such as the main menu and the context menu in Modelica projects view.

3.3.5 New project wizard

The wizard is implemented by the *NewProjectWizard* class. The class is mostly a wrapper around the inner class *NewProjectPage* and *ModelicaCore*'s *createProject()* method. *NewProjectPage* implements the first and only page

of the wizard. This page contains all the widgets for entering information on the new project. The *NewProjectPage* also implements the logic that defines when enough information is entered to be able to create a new project. It also implements checks so that entered information is valid, e.g. that the projects name is unique. When *NewProjectPage* contains valid information of the right amount, the "Finish" button is enabled.

The method *performFinish()* in *NewProjectWizard* class handles the situations where the user decides to go ahead and click on the finish button. It extracts the information from the *NewProjectPage* widgets and forwards it to the *ModelicaCore.createProject()* method. The *createProject()* method handles all the details of creating a Modelica project in the workspace.

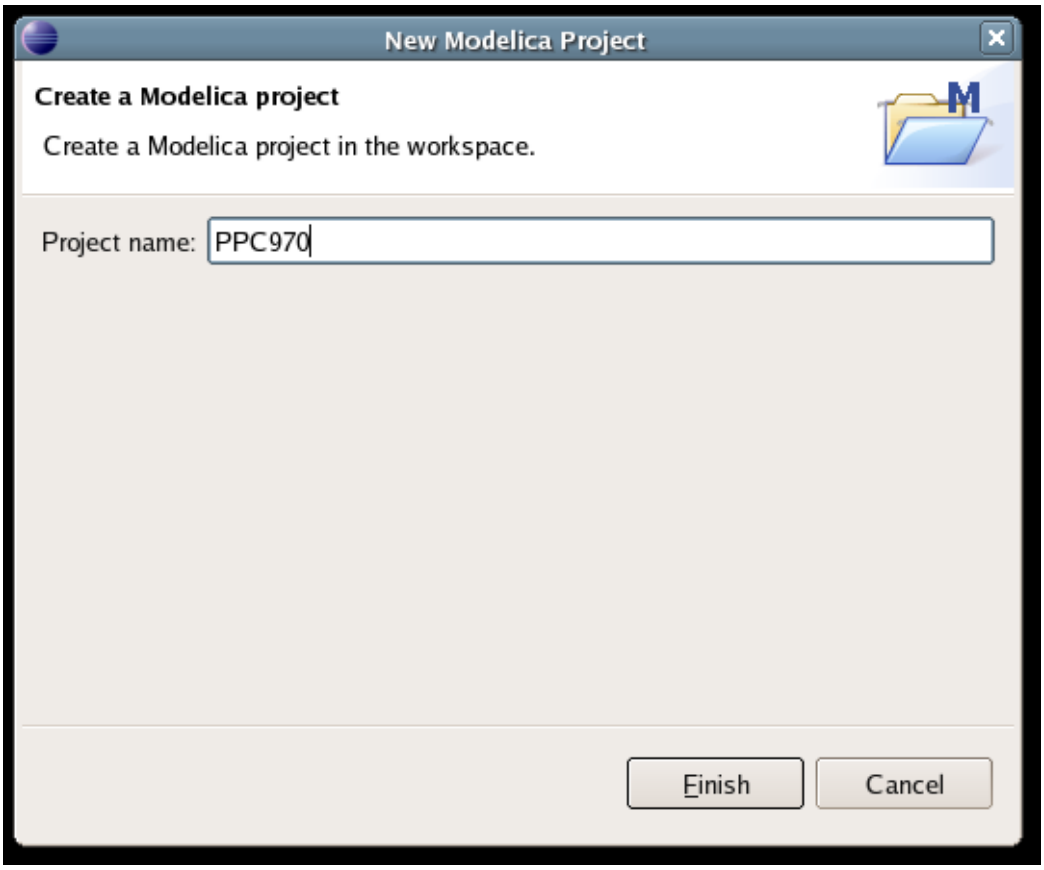

Figure 4: New Modelica Project Wizard

3.3.6 New package wizard

The new package wizard is implemented by the *NewPackageWizard* class. It contains two inner classes, *NewPackagePage* and *PackageCreator*. The *NewPackagePage* implements the only page present in the wizards. The page contains widgets for entering information on the new, soon to be created, Modelica package.

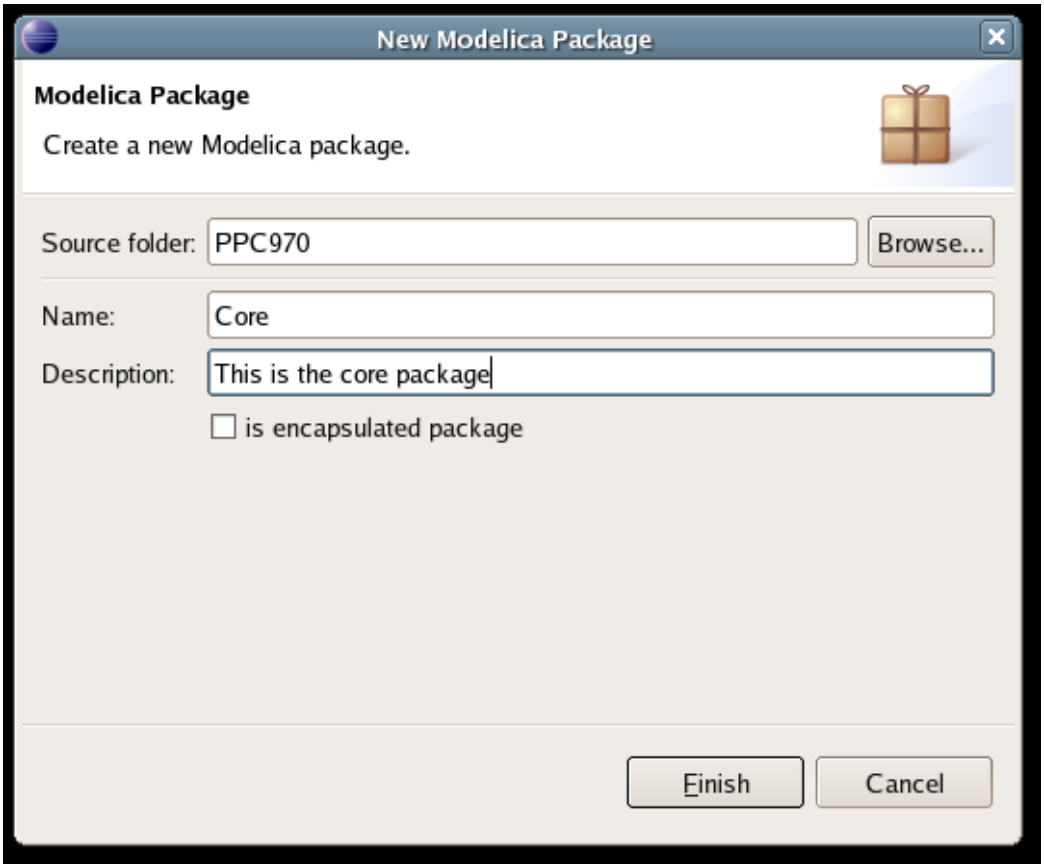

Figure 5: New Modelica Package Wizard

When the wizard is created by the platform, the current selection is passed along. This selection is forwarded to the *NewPackagePage*. If a single Modelica element or file system resource is selected, *NewPackagePage* tries to determine the path of the innermost folder that contains that element. This path is used as a default value in the source folder field.

When the finish button is clicked, information from the fields are harvested and fed to a new instance of the inner class *PackageCreator*. This object is run in a separate thread to avoid blocking the UI queue thread.

PackageCreator creates the new folder for the package, it also creates a *package.mo* file inside that folder with proper definitions.

3.3.7 New class wizard

The wizard is implemented by the *NewClassWizard* class. The class is a 500 lines long jungle, hope you did not forget your machetes! Figure 6 displays the new class wizard. The inner class *NewClassPage* implements the wizards only page. The wizard allows the user to select (a) the restriction type of the class to be created. Based on the restriction type the wizard enables a set of (b) modifiers on the class. For example for a class of restriction type function, the user can select if it will have an external body. Based on the restriction type and modifiers selected, the wizard generates the source code.

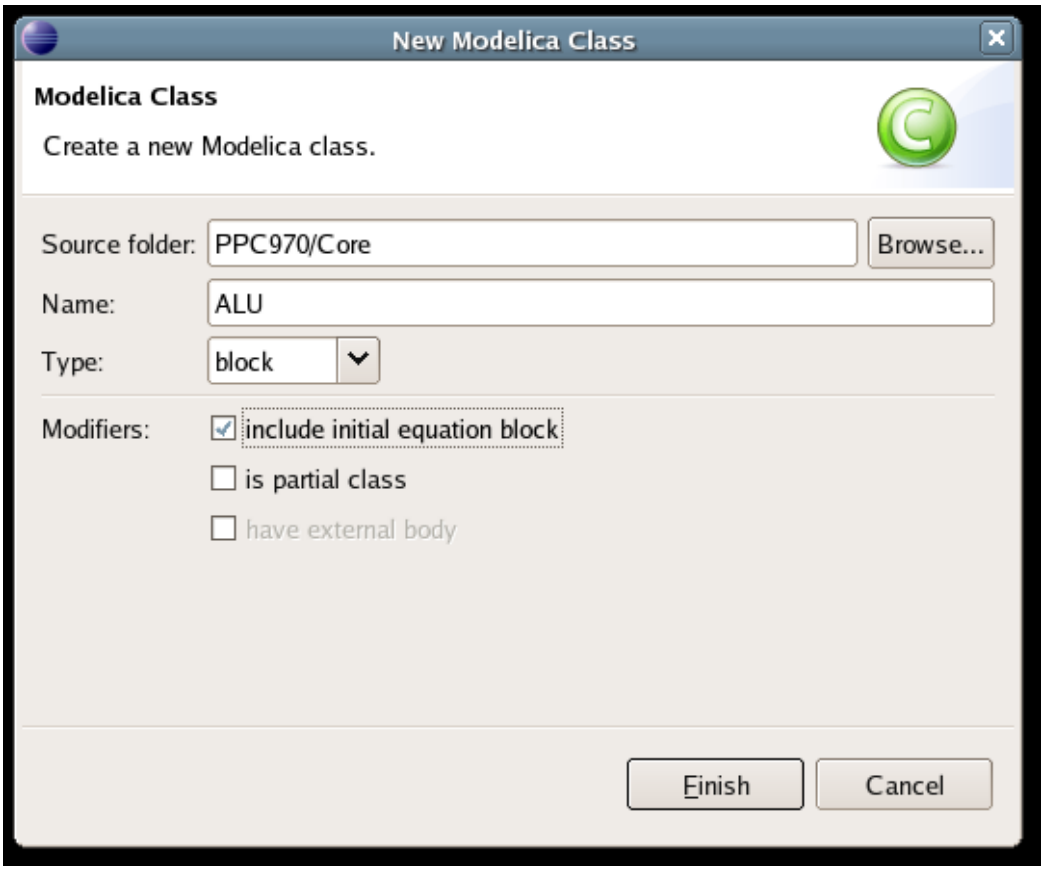

Figure 6: New Modelica Class Wizard

The logic for enabling the modifiers checkboxes is implemented in the *NewClassPage.restrictionTypeChanged()*. The method is invoked from an

anonymous listener on the (a) restriction type widget.

The new class wizard implements the same functionality as the new package wizard, where the current selection is used to determine the default value for the source folder field. Possibly the code should be merged to some abstract class.

The code generation is launched from the *NewClassWizard.performFinish()* method when the user clicks on the finish button. The method gathers information from the widgets and passes it along to the *doFinish()* method as arguments. *doFinish()* is run in a separate thread to avoid blocking the UI thread. Contents of the new source code file are generated by the *generateClassContents()* method, which *doFinish()* invokes. *doFinish()* then creates the file in the specified source folder and writes the generated source code there.

3.4 Error management

The are two primary sources of errors in MDT, the operation on the file system and the Modelica compiler. When designing the code that handles the error conditions important design goals must be kept in mind. The environment should fail gracefully. The user must be notified of the error condition and, most importantly, user created data must not be lost. It is also important to make errors traceable to aid troubleshooting. See also [18] for a discussion on error management in Eclipse plugins.

The general design pattern on error management in MDT is to forward the error condition to the client. On errors in the omc plugin, generally an exception of the subtype of *org.modelica.mdt.compiler.CompilerException* is thrown. This exception is then forwarded via the core plugin to the client, i.e. the ui plugin. When the core plugin interacts with the file system, or some other eclipse runtime services, errors are signalled by throwing the *org.eclipse.core.runtime.CoreException* exception. This exception is generally forwarded to the invoking client.

When the error exceptions reaches the ui plugin we tried to follow the philosophy outlined in [18]. We display errors to the user whenever it is clear it was triggered by some specific user action. In all cases the errors are logged in the Eclipse system log. The code to display errors to the user and write errors to the log is contained in the *org.modelica.mdt.ErrorManager* class.

3.4.1 Logging errors and warnings

In some situations it is not appropriate to show an error message to the user. However it can become tricky to troubleshoot problems if occurred

errors never leave a visible trace. In such situations it is appropriate to log the error to the problems log. It's also possible to write more detailed and more technically formulated error messages, because the average users is not expected to read the problems log. The *ErrorManager* class provides logging facilities in such situations.

3.4.2 Displaying error messages

Whenever the decision is to notify the user about the error, the methods *showCompilerError()* or *showCoreError()* in the ErrorManager class are called. The method called depends on the type of the exception caught. Subtypes of the *CompilerException* class are handled by the *showCompilerError()* method and subtypes of *CoreException* by the *showCoreError()* method.

The *showCompilerError()* and *showCoreError()* methods logs the errors, formulates the error message and displays the message to the user. Both the error message and the log message are formulated by using the type of the exception and any extra information on the error which may be embedded in the exception object. These methods also implements the logic of error notification policy.

3.4.3 Error notification policy (move this to the discussions chapter?)

Displaying the error messages to the user presents an interesting usability dilemma. On one hand the user should not be bombarded with error dialogs. For example while expanding the Modelica node in the standard library sub tree in the projects view, the *showCompilerError()* can be invoked one time for each child under the node if an error condition occures. Depending on the version of the standard library it can be more then 10 times under less than a second. On the other hand the plugin should not fail silently, the user must understand why the computer does not do what it was told to do.

We decided to resolve the dilemma by creating a policy where the number of repeated identical error messages is limited. Errors signalled by exceptions of the types *CommunicationException*, *ConnectException* and *CompilerInstantiationException* are only shown once. If such exceptions are forwarded multiple times to the *ErrorManager.showCompilerError()* they are only logged but no error dialog is shown to the user a second time. The rationale is that the MDT after such errors lacks access to the Modelica compiler, which practically makes it useless. In the face of such errors only thing the user would be interested in is to fix the compiler problem and restart Eclipse.

Errors signalled by exceptions of the types *InvocationError* and *UnexpectedReplyException* have a minimal timeout period between appearances. The timeout is defined by a constant in the *ErrorManager* class and currently is set to 1 minute. These types of errors are of a more transient nature and MDT may as well be fully functional later on. In this case you are mostly interested in avoiding seeing more than one error message from the cluster generated by some specific action.

Errors signalled by the exception of some subtype of *CoreException*, which are handled by the *showCoreError()* method, are always shown to the user and noted in the problem log.

Maybe a more elegant solution can be found to the above outlined usability dilemma. However the authors of this report were not able to arrive at any other workable solutions.

3.4.4 Compiler exceptions

org.modelica.mdt.compiler.CompilerException is the super class of exceptions that signal errors that occurs while communicating or trying to establish a connection to the Modelica compiler.

<figur pa classhierarki av CompilerExceptions>

- *CompilerInstantiationException*

  CompilerInstantiationException is thrown when there was an error instantiating the compiler object specified by the plugin in the declaration of the extension *org.modlica.mdt.compiler*. The exception object's method *getProblemType()* provides more details on the problem encountered. For example, if there was more than one extension defined of the compiler extension point, this exception is thrown.

- *ConnectException*

  ConnectException is thrown when there is an error while trying to establish a connection to the Modelica compiler. This exception can be thrown from many methods due to the fact that connecting to the compiler is implemented lazily. For example if the omc plugin fails to find the compiler binary this exception is thrown.

- *CommunicationException*

  CommunicationException is thrown when there was problems sending a request to the compiler or receiving the reply. This can happen for example if the compiler crashes and dumps core on some particularly nasty request.

- *UnexpectedReplyException*

  The *UnexpectedReplyException* exception is thrown when the compiler replies with something not quite expected. For example if the compiler replies with a string instead of the expected integer. This is typically a sign of compatibility problems with the compiler or bugs in the compiler or the plugin code.

- *InvocationError*

  InvocationError is thrown when the compiler returns an error reply instead of the usual reply. This can happen for example if the method *IModelicaCompiler.loadSourceFile()* is invoked with a path to a non-existent file.

3.5 Bug management

Whenever the MDT code reaches an illegal internal state, for example a point in the code that never should be executed then it is an almost certain sign of a bug. To help troubleshooting, and to help expose less obvious bugs, all such illegal internal states should be logged. The bugs are logged with the *ErrorManager.logBug()* method. Description of the illegal state and location in the source code where it was encountered are written to the log to help tracking down the problem causing the bug. In all locations where it is obvious that an illegal state is reached, a call to *logBug()* should be made.

4 Regression testing

We believe that scripted regression testing is an important tool to improve both the user perceived quality of the product and the maintainability of the code. By making it easy to run the full set of tests more bugs and other issues can be detected early and fixed. A more systematic approach to testing also allows to detect defects which otherwise would be easily overlooked and can be hard to track down and reproduce. If there is a set of regression tests that covers the code it also gives the developer more freedom to refactor that code without fear of introducing new defects. This helps to improve readability and maintainability of the code. For a longer discussion on the role of scripted testing and refactoring in a development process see [11].

4.1 Testing tools

For running the regression tests on MDT two special Eclipse plugins are required, the JUnit PDE plugin and the Abbot for Eclipse plugin.

JUnit PDE is a set of plugins that integrates the unit testing framework JUnit into the Eclipse environment. It allows running the regression tests on plugins from a special instance of Eclipse. The Eclipse SDK package, version 3.0 and later, include all required JUnit plugins to run MDT regression tests. See [4] for more information on creating and running tests with the JUnit framework.

Abbot for Eclipse is the plugin that allows writing scripted tests of GUI components. The plugin mimics the real user input such as key presses and mouse clicks and allows for realistic testing. The Abbot for Eclipse plugin must be installed separately, see [5] for details on obtaining and installing it.

See section 6.2 for a discussion about GUI testing and the tools used in the process.

4.2 Tests plugin project

The regression tests are all placed and run as a separate plugin project *org.modelica.mdt.test*. All tests are implemented as JUnit test cases. Each test class subclasses the *junit.framework.TestCase* class. The test class groups the tests which are run in the same environment. The testing environment is set up by the protected method *setUp()* in each test class. The method is automatically called by the JUnit framework before running the test code. The tests are implemented by the public methods with names that begin with the lower case word *test*, e.g. *testParseList()*.

If a test case class is written to perform test on a specific class a convention exists to name the test case class after the tested class. For example if tests are written for the class *Foo* then the test case class is named *TestFoo*. This convention is intended to help navigate among tests.

4.3 Abbot tags

To direct simulated user input to the widgets in the regression tests you first need to get a reference to the widget. Abbot provides multiple methods to acquire such references. One way is to attach a so called tag on the widget and ask Abbot to fetch the widget by tag. This method is by far the simplest and most predictable. The downside is that it requires support in the code that is tested. We believe that the time saved by being able to write test code faster and with less hassle is worth the extra trouble of modifying the tested code. The tagging of widgets is used whenever it is possible.

Abbot tags are attached to the widgets by setting an attribute "name" to a specific string. Widgets attributes are set by calling the *setData()* method on the widget object. A convenience method, *MdtPlugin.tag()*, is defined to handle the details of tagging widgets. Widgets that regression tests need access to are tagged by the code that creates them. The convention is to define a constant named after the widget in the class that sets the tag. For example the *sourceFolder* widget, in the new class wizard, have the tag constant *SOURCE_FOLDER_TAG*. The constants value is set to the tags value and referred by the regression test code.

In some cases test code need access to a widget created outside of the MDT code. For example the tests on new class wizard need to simulate the click on the wizard's finish button. In such cases more elaborate code needs to be written to acquire the widget reference. Typically such attributes of the widget as caption, type or contents are used to find the desired widget. For example to find the finish button, a widget of type push button with caption 'Finish' is searched for. Here you always run the risk of finding the wrong button if there are two finish buttons displayed simultaneously.

4.4 Utility classes

The package *org.modelica.mdt.test.util* contains the helper classes for writing regression tests. Common code is collected in classes defined in this package.

Many test cases require either a Modelica or a plain project or both to exist in the workspace. To avoid writing the same project creation code in multiple test case classes the class *Area51Projects* was created. The class contains

code to setup two fully populated projects. The *Area51Projects.createProjects()* method creates a Modelica and a plain project. Projects are instantiated with a rich hierarchy of elements suitable to run tests on. The class keeps track of if it has already created the project and avoids trying to create them a second time. This makes it safe to call the *createProjects()* method multiple times from different test cases.

4.5 The untested code

Unfortunately writing regression test took often a backseat due to lack of discipline and external pressures. In particular writing new GUI regression tests was abandoned during the second half of the project. This means that a lot of code is untested and a great deal of bugs undiscovered. See the discussion on GUI testing in section 6.2 for more information. Also the fact that for one line of regression tests code there exists two lines of the product code suggests that there are large areas of untested code. Some of the known white spots on the testing are the GUI code for the Modelica projects view and the Modelica text editor. It is probably a good idea to otain some testing coverage data to know for sure what areas need more testing.

4.6 Tests for known bugs

The ambition while working on MDT was that each time a new bug was discovered to write a regression test that triggers that bug. That ambition has largely gone unfulfilled. However it is noted in the docs/BUGS file whether a regression test exists for the bugs listed there. Also, in the source code comments for the regression tests, it is noted if the test triggers a particular bug.

5 Future work

No program is complete, and MDT is no exception. Below is a list of different parts of MDT that is either missing completely or need to be improved.

5.1 Filtering support

When working with many projects in MDT, you should be able to filter out the projects that you're not interested in at the moment. This should be accomplished by defining filters in the Modelica Projects View. You should for example be able to filter out libraries that are cluttering up the Modelica Projects View.

5.2 Link With Editor

A standard feature in Eclipse is to link the project browser with the editor. This means that when the user selects an editor window among the open editor windows, that document is highlighted in the project browser.

5.3 Standard toolbar

The standard buttons Back, Forward, Up, Collapse All, Link With Editor, and the Filters/Working set menu should all be added to the Modelica Projects View.

5.4 Source code navigation support

JDT allows users to browse the source code in the workspace. For example to see the definition of a method the user can simply press CTRL and click on the function's name anywhere it is used in the source code. The file where the method is defined will be opened in the text editor and the methods body will be made visible.

We belive that this feature is a major time saver while working with any source code of non-trivial size. Such feature should be implemented in MDT sooner rather then later. To be able to do this, OMC must be able to provide more information on the source code structure then it does now. See section 6.1.2 for a detailed discussion on the type information required.

5.5 Quickfixes

Quickfixe should be work like in JDT, where fixes to errors are proposed by the plugin. Quickfixes is a really neat feature and speeds up the workflow considerably. To be able to implement this feature OMC will have to report more detailed error messages. Currently some kind of more structured error management in OMC is being worked on. This can hopefully be used in the future to implement quickfixes.

5.6 Multiple Modelica Compilers

If multiple Modelica Compiler plugins, i.e. plugins that extends the *org.modelica.mdt.compiler* extension, are available, an error message is displayed and nothing works until exactly one compiler is available. This should be changed to allow the user to configure which compiler to use on a per project basis. It will probably be a good idea to add a default compiler setting as well. This is a rather far fetch feature as currently there is no pressing need to use any other compiler besides OMC with MDT.

5.7 Running a simulation

To make a complete environment out of MDT, the simulation of models will have to be supported. This can for example consist of a model setup wizard (where one changes the starting values of the model), a report window with simulation values, and a graphical plot of selected functions. Dymola is an example of how a Modelica simulation environment can look like.

This additional functionality requires that the Modelica Compiler has support for running simulations. OMC has this functionality, so one can add support for simulating models by only modifiying the MDT source code.

5.8 Testing

5.8.1 In general

There are some things that are not tested at all, and these should of course have tests written for them. Some of these are the Modelica Projects View and the Modelica Editor. To get a complete picture of what tests are missing, testing coverage data should be generated.

5.8.2 GUI recording

To create regression tests for the GUI some extra tools should be used. Using a GUI recording tool, rather than write the tests by hand via Abbot, would save a lot of time and hassle. The TPTP suite of tools[1] should be investigated for the presence of a GUI recorder.

5.9 Move Wizards code

The code that creates new packages, new classes, and new projects should be moved from the wizard classes to *org.modelica.mdt.code.internal* and made accessible through a public API in *org.modelica.mdt.core*.

5.10 Split Up NewClassWizard

The NewClassWizard is way too long and complicated. Maybe it should be split into several files.

5.11 Integrated debugger

Another big thing that's missing to make MDT a little more complete is an integrated debugger. The authors are not sure how Modelica is debugged, but they know there has been some work done on creating a stand alone Modelica debugger. This debugger should be investigated before starting work on an embedded debugger in MDT.

6 Discussion and Related work

6.1 Integrating the OpenModelica Compiler

As described earlier the *org.modelica.mdt.omc* plugin provides access to the OpenModelica Compiler (OMC) for the core plugin. At this point MDT mostly needs access to a Modelica parser. By parsing the contents of the Modelica source code files it's possible to implement a number of features. For example browsing definitions in a source code file in projects view is dependant on the ability to parse the file. In the future other OMC features would have to be accessed. For example it would be nice to be able to run simulations directly from the MDT environment. To do that the OMC modules that handle the simulation needs to be accessed.

6.1.1 The OMC access interface

The OMC defines two quite similar ways to access its functionality. The access is available through either a TCP socket or a CORBA-defined interface. We have chosen to use the CORBA interface as described earlier in the Architecture chapter. However there exists some problems with the way that the interface is designed.

The IDL-defined interface only consists of one function, *sendExpression()*, which takes one argument of type string. The function returns a string as well. All communication with OMC is done by sending special text messages via the *sendExpression()* function and parsing the reply, see the description earlier in this report for details on communicating with OMC.

Here a custom text based protocol, on top of CORBAs provided facilities, is defined to access the compiler. For example to retrieve the contents of the "Modelica" package the following actions need to take place. A text expression for querying of class contents must be formulated and passed on to the *sendExpression()* CORBA stub. The stub code marshals the expression into the wire format and sends it over to the server stub function in the OMC. The server stub unmarshalls the received expression to a string and invokes the local implementation of *sendExpression()*. Now OMC needs to parse the received expression before it knows what to do. After the service is performed the reply text must be constructed and sent over the CORBA stack. Once again the reply is marshalled, sent over the wire protocol, unmarshalled and handed over to the MDT code. Now the reply must be parsed to retrieve the contents of the "Modelica" package.

This means that in addition to the CORBA provided marshalling layer a hand written layer needs to be created both in MDT and in OMC. Creating

a custom marshalling layer has a number of drawbacks. First it takes time to write, debug and maintain the marshalling code. It adds extra complexity to the code base, and that's never a good idea, especially when undergraduate students are involved. Also the extra layer consumes additional computing resources such as processor time and volatile storage.

The reason the text based protocol exists is to allow accessing the OMC via a TCP socket. A TCP socket protocol does not define a marshalling layer and thus nessesesestats a custom defined layer. The same text protocol is used for both socket and CORBA based access interfaces.

We think that it is a mistake to provide both access interfaces to OMC. Right now all the drawbacks of both socket and CORBA interfaces are present but none of the advantages. You need a custom design marshalling layer to support the socket layer. However you also have drawback of the overhead and the extra hassle it means to have your code depend on a CORBA package.

Our recommendation on future development of OMC is to drop one of the interfaces. If the support for the socket interface is dropped, the full IDL interface can be defined and enjoy the wonders of an automatic marshalling layer. If the support for CORBA interface is dropped the dependency on the CORBA package will be eliminated and the overhead of double marshalling layers avoided.

Of course, dropped support of one of the interfaces means that backward compatibility will be broken. This would require some work on OMC clients that where using the deprecated protocol. This must be taken into consideration before removing support.

6.1.2 Level of information on parsing

The amount of information the OMC parser provides is not sufficient to implement some features that are desirable to have available in MDT. Features such as refactoring, code navigation directly from the text editor, quickfixes suggestions and others requires to have access to quite detailed information on the source code. Basically you need access to the abstract syntax tree in many cases. Many times you need to know the tokens that are present in the some text area. The type and start and end of the token are needed information. Some work in this area has been discussed among the OMC developers team, and will hopefully be performed in the foreseeable future. Also see section 5 on which possible feature would require what type of extra information from the OMC.

Also, error reporting from the parsing phase should be come more standardized. To implement quickfixes, suggestions in MDT the error messages

should contain such information as severity of the error, type of error, the area in source code where this error is found and so on.

The work on redesigning the error reporting facilities of OMC is done as this report is written. With some luck this section will be obsolete by the time you read this.

6.1.3 Distribution of MDT and OMC

Currently setting up a fully functional MDT environment is quite a lot of hassle. You need to obtain and install the OpenModelica Compiler package. You need to obtain and install the Eclipse SDK package, and finally you need to instruct Eclipse to download the MDT feature from the update site. The amount of trouble required probably scares away a number of potential users. There is at least two ways to streamline the process a bit.

One possibility is to create a single package that bundles OMC, Eclipse and MDT in one single swoop. This type of package would be of interest to users looking for a complete Modelica development environment. The big downside of this solution is the size of the package, which would be at least a couple of hundreds megabytes. Also users who are already using some other third party Eclipse plugins would not like this solution. They would be faced with the alternative of either having two copies of Eclipse installed or trying to merge the two provided distributions by hand.

Another solution would be to create a special Eclipse plugin that would contain the OMC native binary. Such a plugin would be included in the MDT feature. Then setting up the MDT environment would be a two step process, install Eclipse and install the MDT feature. The downside of this solution is that a separate OMC binary plugin for each supported platform is needed. One for Windows, one for each flavour of Linux and so on. Another drawback is that the users who would want to use OMC outside of the Eclipse platform whould need to have two installation of the OMC binaries as the MDT bundled binaries whould not be usable outside of Eclipse.

The above problems whould go away if OMC and MDT whould be distributed on a platform that already ships Eclipse, supports dependencies among packages and provides package repositories. An example of such a platform is Fedora Core 4 which is built around an rpm packaging system and supports yum repositories. MDT could be destributed in the following way. OMC is packaged as an rpm. MDT is packaged as a second rpm which specifies a dependency on Eclipse and OMC packages. Both OMC and MDT are uploaded to a yum repository, where Eclipse is is already bundled with Fedora Core.

Installing a fully functional MDT would be as easy as typing 'yum install

mdt'. Unfortunately this is not possible at the moment because MDT would not run on the free Java implementation bundled with Fedora. Distributing the proprioretary Java implementation in a user friendly way with Fedora Core is not possible due to licensing restrictions.

Anyway this report is not trying to solve the worlds packaging problems, this is some one else's work.

6.2 Testing of GUI code

Writing regression tests for the GUI code turned out to be quite a bit more problematic then expected.

The first problem was to find a library that provides hooks for simulating user input for SWT. Information available on the Internet on how to do GUI testing with SWT and in particular in the Eclipse environment is hard to come by. There seems to exist two libraries, Abbot for Eclipse and TPTP[1].

We chose Abbot mostly because we did not manage to figure out for sure that TPTP provides such functionality. As a matter of fact we are still not quite sure, however we managed to secure a copy of a TPTP user manual.

Abbot basically lacks any sort of documentation besides the partial API documentation and some bits of outdated tutorials and code examples. However, on the plus side, it's not too hard to figure out how to use Abbot even with the small scraps of information available. The API is pretty much straight forward. The existing regression tests are probably useful illustrations. Tests on wizards are good examples to look at.

Besides troubles with learning to use the Abbot library, writing GUI regression tests turned out to be quite complex and time consuming work. While doing GUI testing the threading model of the toolkit must be considered. There exists some rules on actions that must be done on and off the thread that processes the GUI event queue, see [8] for more information. Also, due to the threading model of SWT, some tests must run in multiple threads which need to synchronize with each other.

Due to the complex nature of writing GUI tests and due to the fact that the MDT development team lack solid understanding of threading issues at hand, writing new GUI tests was halted during later stages of the project. Our recommendation to the future MDT developers is to gain a solid understanding of the SWT and Eclipse threading model early in the project.

Writing the GUI tests by hand should probably be avoided. Probably there exist some GUI recording tools that can be employed instead. Such a tool can save a lot of time. We recommend to take a close look at the TPTP project and consider migrating current regression tests to it. To stop using Abbot is probably good idea. It would certainly be nice to remove all the

abbot widget tags, look for string constants who's name ends with _TAG in GUI-classes.

6.3 Modelica compiler interface

The interface that the core plugin uses for accessing the Modelica compiler is quite OMC-centric. The methods defined in the *IModelicaCompiler* interface mirrors quite closely the functionality provided by OMC's interactive API. It relies heavily on the concept of a memory database of Modelica elements. The interface assumes that elements are loaded into the compiler's memory from files and that the compiler later on can be queried on the contents of the database.

The errors that can be signaled out by the interface also makes assumptions based on the way OMC works. For example the "unexpected reply" error assumes that communication with the compiler is done via a text based protocol.

All these assumptions makes it hard to add support for other Modelica compilers. The interface should be reworked if support for some other compiler than OMC is needed. However, currently there is no need to support any other compilers, so this task is not of pressing nature. Also making a more general interface could be tricky and cause inefficient code.

7 Conclusions

7.1 Accomplishments

In the original thesis proposal, a complete IDE with refactoring and debugging support should have been made. This has not been accomplished, but what we've accomplished is hopefully a start for a complete IDE for Modelica development.

MDT allows you to:

- Edit Modelica files with an editor that has syntax highlighting.
- Discover syntax errors in files that you're editing.
- Browse the package and class hierarchies that your project contains.
- Browse the Modelica Standard Library and inspect the source code.
- Type code faster by utilizing code completion.

Many of these feature depend on the OpenModelica Compiler, and some features that we didn't implement (for example refactoring) depend on features that are missing in OMC. See the discussion about the CORBA interface in section 6.1.1.

7.2 What we deliver

As is always the case with the development of a relatively complex system, alot of artifacts have been produced. Below is a little detail about the various parts that we deliver.

7.2.1 The plugins

The three plugins (core, ui, omc) that we provide can be easily fetched by using the provided update site[9]. You can visit the update site with a web browser to get instructions on how to use the update site from Eclipse.

7.2.2 Documentation

We provide two kinds of documentation, the user manual and the development documents. The user manual can be reached from within Eclipse when MDT is loaded by simply selecting Help Contents from the menu item Help. From the Help Contents page you can reach the Modelica Development User Guide.

The development documents are located in the MDT repository on some subversion server. The location of this repository is secret right now (i.e. we're not sure where it will be).

7.2.3 Source code

The source code of MDT is available in the MDT repository. As already stated, we don't know where it is. I mean, it's a secret. Really.

References

[1] The eclipse test & performance tools platform website. `http://www.eclipse.org/tptp/`.

[2] Eclipse website. `http://www.eclipse.org`.

[3] Free software definition. `http://www.gnu.org/philosophy/free-sw.html`.

[4] Junit website. `http://www.junit.org/index.htm`.

[5] Mdt hacking manual. located inside the MDT source code repository, `docs/HACKING`.

[6] Modelica website. `http://www.modelica.org`.

[7] Openmodelica users guide. Included with the OpenModelica Compiler package, `http://www.ida.liu.se/~pelab/modelica/OpenModelica/`.

[8] Swt threading issues. `http://help.eclipse.org/help31/index.jsp?topic=/org.eclipse.platform.doc.isv/guide/swt_threading.htm`.

[9] Update site. `http://www.ida.liu.se/~pelab/modelica/OpenModelica/MDT/`.

[10] Modelica - a unified object-oriented language for physical systems modeling, tutorial. 2000. `http://www.modelica.org/documents/ModelicaTutorial14.pdf`.

[11] Kent Beck and Cynthia Anders. *Extreme Programming Explained : Embrace Change*. Addison-Wesley Professional, 2004.

[12] Mihaly Csikszentmihalyi. *Flow: The Psychology of Optimal Experience*. Harper Perennial, 1991.

[13] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. IEEE Press, 2004.

[14] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*, chapter 10.3.3.2. IEEE Press, 2004.

[15] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*, chapter 3.11.1. IEEE Press, 2004.

[16] Erich Gamma and Kent Beck. *Contributing to Eclipse*. Addison-Wesley, 2004.

[17] Erich Gamma and Kent Beck. *Contributing to Eclipse*, chapter 31 Core Runtime - IAdaptable. Addison-Wesley, 2004.

[18] Erich Gamma and Kent Beck. *Contributing to Eclipse*, chapter 20 Exceptions Handling. Addison-Wesley, 2004.