Modelica Development Tooling for Eclipse

Elmir Jagudin
Andreas Remar

April 10, 2006

LITH-IDA-EX–06/024–SE

Summary

PELAB at Linköping University is developing a complete environment for developing software in the Modelica language. Modelica is a computer language for modelling and simulating complex systems that can be described by using mathematical equations. The language is currently being extended by PELAB to also support language modeling and meta-modeling. The environment that PELAB is developing is called OpenModelica.

An important part of this project is a textual environment for creating and editing Modelica models. The Modelica Development Tooling is a collection of integrated tools for working with Modelica projects. It is developed as a series of plugins to the Eclipse environment.

One of the requirements of an integrated development environment (IDE) is to provide the user with extra information that is helpful for developing and understanding software. Some of the information provided by MDT is package browsing and error marking. Modelica packages can be browsed in much the same way as Java packages can be browsed in JDT, the Java Development Tooling. JDT is in fact a source of inspiration for MDT. That should be obvious given the name.

Some of the functionality that is provided by MDT is provided by the OpenModelica Compiler. This compiler is a part of the OpenModelica Environment developed at PELAB. It has support for storing loaded models in memory and allowing queries to be performed on the model at runtime. This is utilized in MDT to allow the Project Browser to display the classes contained in Modelica files, and also to provide MDT with code completion proposals.

In summary, MDT allows you to:

- Edit Modelica files with an editor that provides syntax highlighting.
- Discover syntax errors in files that you're editing, and provide helpful ways to find where these errors are located.
- 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.

Keywords

Modelica, Integrated Development Environment, Eclipse, OpenModelica

Contents

1 Introduction **1**
- 1.1 Intended Audience . . . 1
- 1.2 Thesis Contributions . . . 2
- 1.3 Thesis Outline . . . 2

2 Background **3**
- 2.1 Software Development Environments . . . 3
  - 2.1.1 Language-centered Environments . . . 3
  - 2.1.2 Structure-oriented Environments . . . 4
  - 2.1.3 Toolkit Environments . . . 4
  - 2.1.4 Method-based Environments . . . 5
- 2.2 Modelica . . . 5
- 2.3 Eclipse . . . 6
  - 2.3.1 Eclipse History . . . 6
  - 2.3.2 Eclipse Platform Architecture . . . 7
- 2.4 OpenModelica Compiler . . . 9

3 Overview **10**
- 3.1 Modelica Development Users Guide . . . 10
  - 3.1.1 Features . . . 10
  - 3.1.2 Known Issues . . . 10
  - 3.1.3 Requirements . . . 11
- 3.2 Getting Started . . . 11
  - 3.2.1 Configuring the OpenModelica Compiler . . . 11
  - 3.2.2 Accessing the Modelica Perspective . . . 11
  - 3.2.3 The Modelica Wizards . . . 11
  - 3.2.4 Making a Project . . . 11
  - 3.2.5 Importing a Project . . . 12
  - 3.2.6 Creating a Package . . . 12
  - 3.2.7 Creating a Class . . . 12
  - 3.2.8 Syntax Checking . . . 13
  - 3.2.9 Code Completion . . . 13
- 3.3 Information Popups . . . 14
- 3.4 Building Modelica Projects . . . 14
  - 3.4.1 Creating a Builder . . . 15

4 Architecture **19**
- 4.1 org.modelica.mdt.core . . . 19
  - 4.1.1 Modelica Elements Access Layer . . . 20

4.1.2 Modelica Projects . . . 20
4.1.3 Folders . . . 20
4.1.4 Source Files . . . 20
4.1.5 Plain Files . . . 21
4.1.6 Classes . . . 21
4.1.7 Components . . . 21
4.1.8 Imports . . . 21
4.1.9 Function Signatures . . . 22
4.1.10 Mapping to the Source Code . . . 22
4.1.11 Reversed Mapping from the Source Code . . . 22
4.1.12 Modelica Standard Library . . . 23
4.1.13 Tracking Element Changes . . . 23
4.1.14 Compiler Extension Point . . . 23
4.2 org.modelica.mdt.omc . . . 24
4.2.1 Communicating with OMC . . . 24
4.2.2 Starting OMC . . . 24
4.2.3 OMC Interactive API . . . 25
4.2.4 OMC Communication Parser . . . 25
4.2.5 Interfacing with the `core` Plugin . . . 25
4.3 org.modelica.mdt.ui . . . 26
4.3.1 Modelica Development User Interface . . . 26
4.3.2 Modelica Projects Browser . . . 27
4.3.3 Opening Elements in an Editor . . . 28
4.3.4 Wizards . . . 29
4.3.5 New Project Wizard . . . 29
4.3.6 Abstract New Type Page . . . 29
4.3.7 New Package Wizard . . . 30
4.3.8 New Class Wizard . . . 30
4.4 Error Management . . . 31
4.4.1 Logging Errors and Warnings . . . 31
4.4.2 Displaying Error Messages . . . 31
4.4.3 Error Notification Policy . . . 32
4.4.4 Compiler Exceptions . . . 33
4.5 Bug Management . . . 34

5 Regression Testing of MDT **35**
5.1 Testing Tools . . . 35
5.2 Tests Plugin Project . . . 35
5.3 Abbot Tags . . . 36
5.4 Utility Classes . . . 36
5.5 Untested Code . . . 37

5.6 Tests for Known Bugs . . . 37

6 Future Work **38**
6.1 Filtering Support . . . 38
6.2 Link With Editor . . . 38
6.3 Standard Toolbar . . . 38
6.4 Source Code Navigation Support . . . 38
6.5 Quickfixes . . . 39
6.6 Multiple Modelica Compilers . . . 39
6.7 Running a Simulation . . . 39
6.8 Testing . . . 39
6.8.1 In General . . . 39
6.8.2 GUI Recording . . . 40
6.9 Move Wizards Code . . . 40
6.10 Integrated Debugger . . . 40

7 Discussion and Related work **41**
7.1 Integrating the OpenModelica Compiler . . . 41
7.1.1 The OMC Access Interface . . . 41
7.1.2 Level of Information on Parsing . . . 42
7.1.3 Distribution of MDT and OMC . . . 43
7.2 Testing of GUI Code . . . 44
7.3 Modelica Compiler Interface . . . 45
7.4 The Modelica Package Structure . . . 45
7.5 Other Modelica Development Environments . . . 46
7.5.1 Dymola . . . 46
7.5.2 MathModelica . . . 46
7.5.3 Free Modelica Editor . . . 47
7.5.4 Modelica Mode for GNU Emacs . . . 48
7.5.5 SciTE with Modelica Mode . . . 49
7.5.6 UltraEdit with Modelica Keywords . . . 50

8 Conclusions **52**
8.1 Accomplishments . . . 52
8.2 What We Deliver . . . 52
8.2.1 The Plugins . . . 52
8.2.2 Documentation . . . 52
8.2.3 Source Code . . . 53

A Package Overview **54**

1 Introduction

The creation of software is a complex and error prone task. To help the programmers, tools have been developed that assist with the developing 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 can be seen as a collection of development tools glued into one large program, so that they can be reached easily. This definition is a bit oversimplified, see section 2.1 for a longer discussion on development environments. 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 this thesis had quite strict requirements on what environment to develop in, i.e. Eclipse, Visual Studio was 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[6]. 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 software.

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.

The Modelica Development Tooling, or MDT for short, is a collection of plugins for Eclipse that provides an environment for working with Modelica software. 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

To get the most out of this report on the Modelica Development Tooling, the reader should have some understanding of how plugins for Eclipse are imple-

mented. Familiarity with the Modelica and Java languages is recommended. Some CORBA terminology is used towards the end of the report. To get a feel for how MDT works, the reader is encouraged to install MDT and read the user manual. See section 8.2 on page 52 for details about how to install MDT.

Contributing to Eclipse[28] is a good introduction on writing plugins for Eclipse. A bit extensive but thorough work on the Modelica language is Peter Fritzson's book[26]. Chapter two "A Quick Tour Of Modelica" is sufficient reading for understanding this text. As far as the authors know, no good books exist on CORBA.

1.2 Thesis Contributions

This thesis gives an overview of the Modelica Development Tooling for Eclipse that has been developed by the authors of this report with some help from Adrian Pop and other members of PELAB. It also contains a comparison of MDT and other Modelica programming environments.

1.3 Thesis Outline

The following is a short outline of the thesis:

- Chapter 2 starts off with a short background of software development environments, Modelica, Eclipse, and the OpenModelica Compiler.
- Chapter 3 contains the Modelica Development Users Guide to get you started using MDT.
- Chapter 4 has a quite detailed description of the architecture of the Modelica Development Tooling plugins.
- Chapter 5 has a discussion about the testing framework and testing tools that were used when developing MDT.
- Chapter 6 contains some suggestions on future work.
- Chapter 7 has discussions regarding the OpenModelica Compiler, the testing of GUI code, and the Modelica Compiler interface. A short walkthrough of other Modelica environments is also presented.
- Chapter 8 concludes this thesis by detailing what we've accomplished and what we deliver.

2 Background

This chapter provides a short overview of general software development environments, the modeling language Modelica, the extensible development environment Eclipse, and the OpenModelica compiler.

2.1 Software Development Environments

An "environment" refers to the collection of hardware and software tools a system developer uses to build software systems. A "software development environment" is an environment that augments or automates *all* the tasks comprising the software development cycle, including configuration management, and project and team management.[25]

To summarize some of the technological trends, the following is a list of different categories of software development environments. A particular environment may fit into many of these categories, as these categories are not competing viewpoints.

2.1.1 Language-centered Environments

A language-centered environment is such an environment that is specially built to support a particular language. Examples of language-centered environments are Interlisp, Cedar, Smalltalk and the Rational Environment.[25]

Environments in the language-centered category encourage an exploratory style of programming. The programming environment and the runtime environment are the same. Code can be developed and executed interactively, both in a top-down and bottom-up style. This allows for rapid prototyping.

Language-centered environments support the exploratory, interactive mode of programming by making available semantic information. Semantic information is typically symbol-table information such as information about the definition and use of variables and procedures and information about types. This information is made available through, for example, browsers. Browsing involves navigating through the set of program objects and their relationships.

Browsers are very useful tools for exploratory software development. They also have the potential of being very effective during program maintenance. As the maintainers often are not the original developers, they often spend a considerable amount of time browsing the code. Browsers help maintainers to determine the scope of a change.

2.1.2 Structure-oriented Environments

A structure-oriented environment is a tool that lets the developer to enter a program in terms of language constructs. The editor is the central component of such environments. It is the interface through which all program structures are manipulated.[25]

Structure-oriented environments have made alot of contributions to environment technology; direct manipulation of program structure, multiple views of the program structure, incremental checking of static semantics, and making semantic information available to the user.

In a structure-oriented environment there are several ways of entering and manipulating structures. One involves only structural editing, it can be seen as template-driven editing. The user selects programming constructs from menus, and the user can not enter syntactically incorrect programs. This menu-driven entering of code is quite cumbersome, so some environments represents expressions as text.

Another approach is a mixed-mode operation, where the user either can enter text as in a normal editor or work directly on the program structure through a structure editor. The user enters program fragments as text and asks the environment to complete the processing. The environment keeps track of both the textual and structural representations of the code, and keeps them consistently updated.

2.1.3 Toolkit Environments

A toolkit environment is a collection of small tools that are intended to support the coding phase of software development. The approach is to start with the operating system and add tools such as compiler, editor, assembler, linker, and debugger. There are also tools for large-scale development tasks such as version control and configuration management. The toolkit approach is language-independent.[25]

Unix is an operating system that has encouraged extensions in the form of programs. The data model of Unix is the ASCII byte stream. By using this data model, different tools in the Unix environment can communicate. Each tool must parse the text stream to extract a structured representation of the data.

Toolkit environments provide tools for working with large-scale software development that are independent of a particular programming language. These tools help the programmers by recording version numbers for source code. Some systems that help with versioning of source code are CVS, Subversion, and GNU Arch.

2.1.4 Method-based Environments

Method-based environments support particular methods for developing software. There are two kinds of classes that these environments fall into: development methods for particular phases in the software development cycle, and methods for managing the development process.[25]

Development methods address various steps in the software development lifecycle such as design, specification, validation, and verification. Different methods has different degrees of formality. One method may be informal, as in written natural text. Another method may be semiformal, as in both written and graphical representations that has limited verifiability. And finally, a method can be formal, with an underlying theoretical model. When using a formal method, there are ways to verify that a description is correct.

Managing the software development process consists of both managing the product under development and managing the process for developing and maintaining the product. Support for product management includes facilities for version, configuration, and release management. Support for managing the development process includes facilities for project management, task management, communication management, and process modeling.

2.2 Modelica

It is often convenient to describe systems with equations instead of using the algorithmic approach when doing simulations. Modelica was developed with this fact in mind. Modelica[13][26] is an object-oriented simulation language for modeling systems that can be described using differential and algebraic equations. 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.
- At PELAB, Modelica is being extended with meta-modeling capabilities, allowing languages such as Modelica being modeled in Modelica[27].

See chapter 2 of Principles of Object-Oriented Modeling and Simulation with Modelica 2.1[26] for an introduction to Modelica model development. This chapter is also available online at the OpenModelica website[18].

2.3 Eclipse

Eclipse[4] is an open source framework for creating extensible integrated development environments (IDEs). The integration simplifies development and avoids disturbing the "flow"[24] that a programmer can attain when programming.

The goal of the Eclipse platform is to avoid duplicating common code that is needed to implement a powerful environment for development of software. By allowing third parties to easily extend the platform via the plugin concept, the ammount of new code that needs to be written is decreased. The leverage of an existing code base decreases time-to-market and creates new synergies in the software tools ecosystem.

2.3.1 Eclipse History

In the mid 1990s the software developments tools were dominated by systems built around two technologies. A lot of of the tools were focused on runtime environment developed and controlled by the Microsoft corporation. The other was build around the Java platform. The Java platform is less controlled by a single company and more open to industry and community input.

IBM felt it was important to contribute to the growth of the more open Java platform to avoid losing business to the Microsoft corporation. By creating a common platform for development tools built on top of the Java platform, IBM hoped to win over more developers from competing systems.

In late 1998, the software division at the IBM corporation began working on the software project that is today known as Eclipse. The original work was based on resources developed by Object Technology International labs. In the beginning, work on a new Java IDE was done at the labs. At the same time additional teams were setup by IBM to build other product on top of the platform.

IBM knew that to achieve broad adaptation of the Eclipse platform in the industry, third parties must be involved in the project. However, other organizations were reluctant to invest resurces in the new unproven technology. In order to increase the rate of adaptation of the platform and to instill confidence in the Eclipse platform, IBM decided to release the code base under an open source license, and to build a community around the project.

In 2001, IBM together with eight other organizations created an Eclipse consortium. A website at eclipse.org was started in order to create and coordinate a community around Eclipse. The goal was that source code would be controlled and developed by the open source community and the consortium would handle the marketing and business side of the project. At that point, IBM was the largest contributor to both the open source community and the consortium.

Two years later the first major public release of the Eclipse platform was made. The release got a lot of attention from developers and was well-received. However, industry analysts suggested that many were still perceiving Eclipse as an IBM-controlled technology. Many key players in the industry did not want to make commitments to a project controlled by the International Business Machines corporation.

After discussions within the consortium it was decided that a new organization was needed to make the status of Eclipse as an open and community driven project clear. At the EclipseCon 2004 gathering an announcement was made that the Eclipse Foundation was formed. The foundation is an independent not-for-profit organization. It has its own full time paid professional staff, supported by foundation members.

The new organization have proven itself a success. At this point the foundation have released version 3.0 and 3.1 of Eclipse since its birth. These releases have gained more adaptation and recognition than any earliers versions. Today the foundation has more than 90 full-time developers on the pay roll and receives more than $2 millions in funding each year.

Currently there are more than eighty member companies in the fundation of which at least sixty-nine are providing add-on products to Eclipse. Today there exists hundreds of proprietary and an even greater number of free plugin products. Eclipse has gained a solid foothold in the industry and is one of the major open source software development platforms.[1]

2.3.2 Eclipse Platform Architecture

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

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.

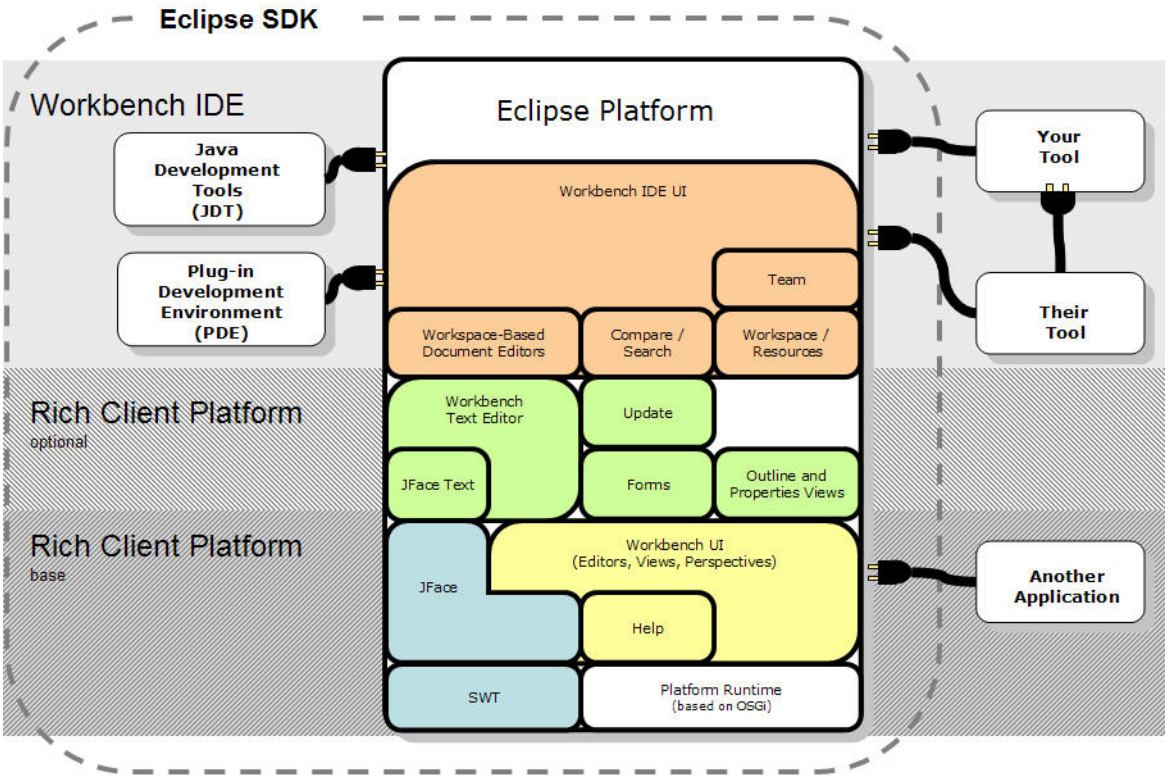

Figure 1: Eclipse Platform Architecture

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 name extension that determines what editor that should be used.

As the number of plugins in Eclipse can be very large, 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[22] is for example provided by the MDT Development Team for easy installation of the latest version of MDT.

2.4 OpenModelica Compiler

The OpenModelica Compiler (OMC) is being developed at PELAB. It is a part of an effort to produce a complete Modelica environment[17] 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 4.2 on page 24. 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.

3 Overview

To get you started with the Modelica Development Tooling, this chapter is a short overview of what MDT provides. The Modelica Development Users Guide can also be reached from Eclipse when MDT is installed by going through the menus *Help > Help Contents*. A browser will open where you will be able to select the Modelica Development Users Guide. The guide you can find there will hopefully describe the most recent version of MDT.

3.1 Modelica Development Users Guide

The Modelica Development Tooling (MDT) Plug-In integrates the OpenModelica Compiler with Eclipse. MDT, together with the OpenModelica Compiler provides an environment for working with Modelica projects.

3.1.1 Features

MDT is still in an early stage of development but already provides a useful environment for the Modelica programmer. The following features are implemented so far:

- browsing support for Modelica classes
- syntax color highlighting
- syntax checking
- syntax error highlighting
- code completion
- wizards for creation of projects, packages, and classes
- Modelica System Library browsing support

3.1.2 Known Issues

The following are some know issues with MDT:

- Due to compatibility problems between the Eclipse Framework and the OMC parser, tab characters in the source code are not supported. This problem arises because Eclipse sees a tab character as a single character, whereas OMC sees a tab character as 8 characters. This leads to a conflict when trying to find out where in a document something

resides. When tabs are present in the sourcecode some MDT features, such as error markers and code completion, will not work properly. Avoid using tabs when possible.

3.1.3 Requirements

MDT requires at least the OpenModelica compiler version 1.3.2, Eclipse 3.1, and Java 1.5. Other versions of Eclipse may work, but have not been tested.

3.2 Getting Started

3.2.1 Configuring the OpenModelica Compiler

MDT needs to be able to locate the binary of the OpenModelica Compiler. It uses the environment variable OPENMODELICAHOME to do so. If you have problems using MDT make sure that OPENMODELICAHOME is set to the folder where the OpenModelica Compiler is installed. In other words, OPENMODELICAHOME should point to a folder where there is a subfolder named "bin" that contains the OpenModelica Compiler binary. This binary is named *omc.exe* on Windows platforms and *omc* on Unix platforms.

3.2.2 Accessing the Modelica Perspective

A *perspective* is a collection of *views* that are useful when developing software. A *view* can for example be an editor frame, a project browser or a list of problems. To access the *Modelica perspective*, choose the *Window* menu item, pick *Open Perspective* followed by *Other...* Select the *Modelica* option from the dialog presented and click *OK*.

3.2.3 The Modelica Wizards

The following sections describes the *wizards* that are a part of MDT. A *wizard* is a user dialog that automates a repetitive and complex task.

3.2.4 Making a Project

To start a new project, use the *New Modelica Project Wizard*. It is accessible through *File* > *New* > *Modelica Project*. After creating a project you can add files and folders to the project by selecting the corresponding wizard found in the *File* > *New* menu subsection. Files having an extension of .mo will be treated as Modelica source code files by MDT. See Figure 2 on page 12.

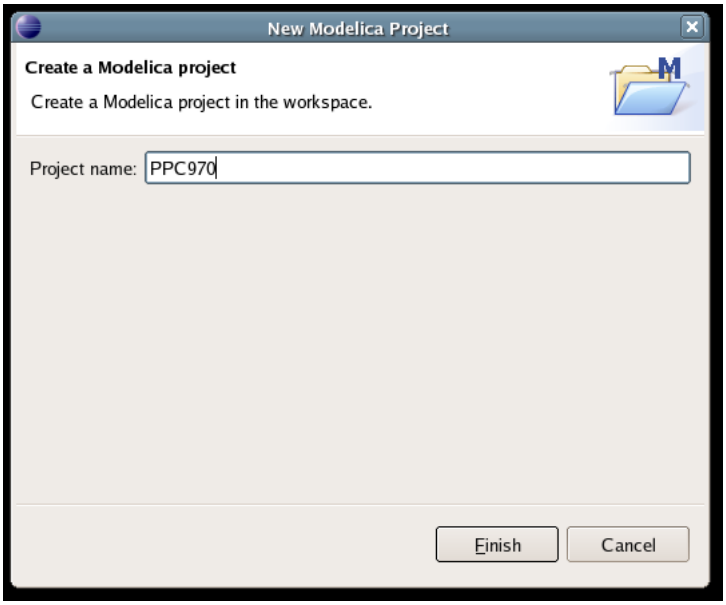

Figure 2: Creating a new Modelica project

3.2.5 Importing a Project

To import an existing Modelica project you need to create an empty Modelica project and populate it with existing files. Create a new Modelica project with the wizard. Use the file system import wizard on that project to copy the files to the project's folder. The import wizard is available by selecting *File* > *Import...* with a Modelica project selected.

3.2.6 Creating a Package

To create a new Modelica package, use the *New Modelica Package Wizard*. You can access it by going through *File* > *New* > *Modelica Package* or by right-clicking in a project and selecting *New* > *Modelica Package*. Enter the desired name of the package and a description about what it contains. See Figure 3 on page 13.

3.2.7 Creating a Class

To make a new Modelica class, select where in the hierarchy that you want to add your new class and select *File* > *New* > *Modelica Class*. When creating a Modelica class you can add different restrictions on what the class can contain. These can for example be *model*, *connector*, *block*, *record*, or *function*. When you have selected your desired class restriction type, you

Figure 3: Creating a new Modelica package

can select modifiers that add code blocks to the generated code. *Include initial code block* will for example add the line *initial equation* to the class. See Figure 4 on page 14.

3.2.8 Syntax Checking

Whenever a Modelica (.mo) file is saved in the Modelica Editor, it is checked for syntactical errors. Any errors that are found are added to the *Problems view* and also marked in the source code editor. Errors are marked in the editor as a red circle with a white cross, a squiggly red line under the problematic construct, and as a red marker in the right-hand side of the editor. If you want to reach the problem, you can either click the item in the Problems view or select the red box in the right-hand side of the editor. See Figure 5 on page 15.

3.2.9 Code Completion

MDT will try to help you with writing code. Code completion starts at strategic positions in the code (when you type a dot (.)) or when you type Ctrl+Space. If you for example type `Modelica.`, a list of the packages and classes that are available in the `Modelica` package will be displayed. See Figure 6 on page 16 to see how this looks like. You can narrow down the matches by typing in the first characters of the class or package that you

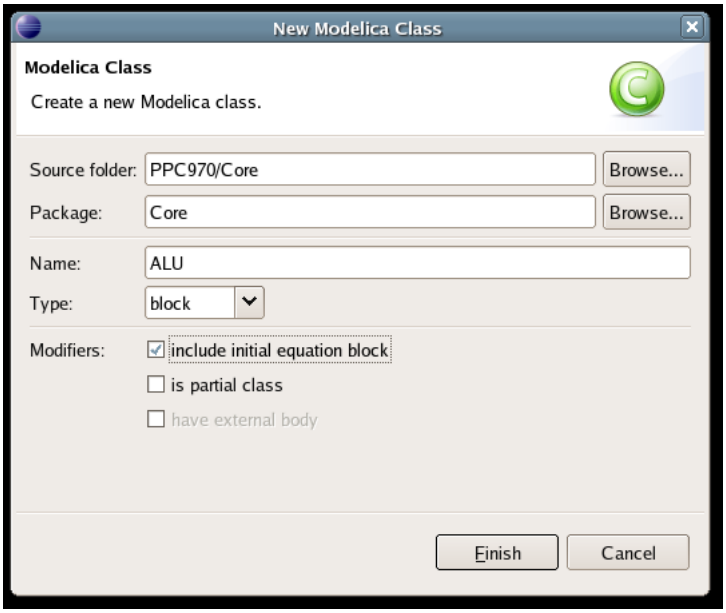

Figure 4: Creating a new Modelica class

want to type in. Thus, if you've first typed `Modelica.` you can then continue typing `Me` and MDT will propose `Mechanics` and `Media` as completions. You can always backtrack the narrowing by erasing characters.

Code completion uses imported packages and the *Modelica Standard Library* when figuring out what code completion to propose. Make sure that you save (Ctrl+S) when you've typed an `import` statement, as then that import will be available when completions are proposed.

3.3 Information Popups

Whenever MDT thinks you're starting to type the arguments to a function call, the system will try to help you by displaying the types of the parameters. See Figure 7 on page 16.

3.4 Building Modelica Projects

Building Modelica projects, e.g. translating the Modelica files to machine code to run simulations, is not supported out of the box right now. However, it is possible to instruct Eclipse to use an external program for building. This is done by creating a so called *project builder*.

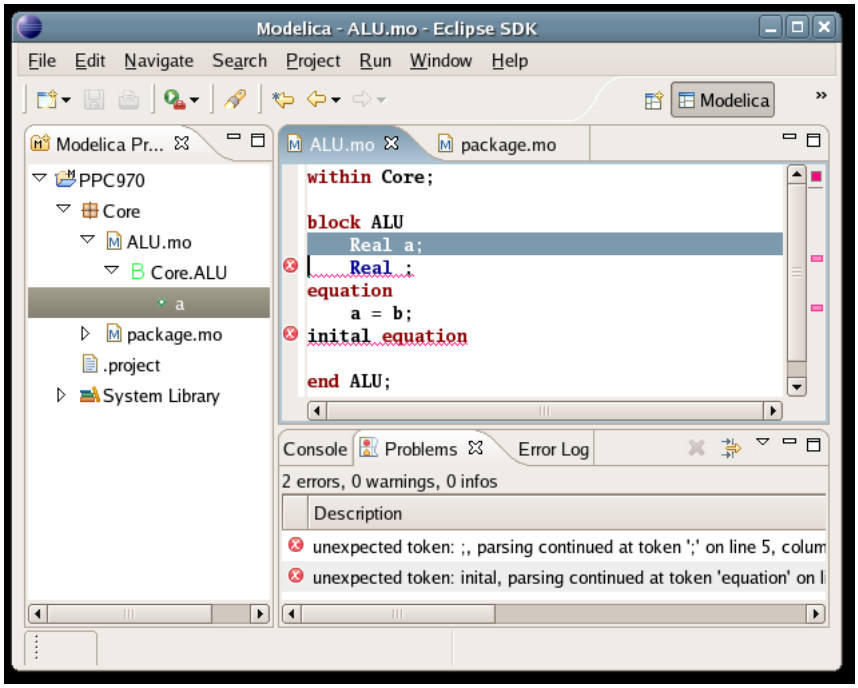

Figure 5: Syntax errors

3.4.1 Creating a Builder

To create a builder, select a project and select *Project* > *Properties*. Select the *Builders* option and then click *New...* Choose *Program* as configuration type. On the next screen, enter the path to the make binary in the *Location* field. To select the *Working Directory*, click *Browse Workspace...* and select your project from the list. See Figure 8 on page 17.

If you're building the OpenModelica Compiler you will also need to set up some environment variables. Click on the *Environment* tab to view the current variables. To add a new variable, simply click *New...* The variables that are needed for building OpenModelica are *ANTLRHOME*, *CLASSPATH* and *RMLHOME*. Refer to the OpenModelica README file for details. See Figure 9 on page 18 for an illustration.

When you're finished, click *OK* and close the *Properties* dialog. Now you have to disable the automatic build feature by deselecting *Build Automatically* in the *Project* menu item. See Figure 10 on page 18.

Now you can build the project by selecting the project and clicking *Project* > *Build Project*. Automatic incremental building of Modelica project executables is not supported at this time, but may be added in the future.

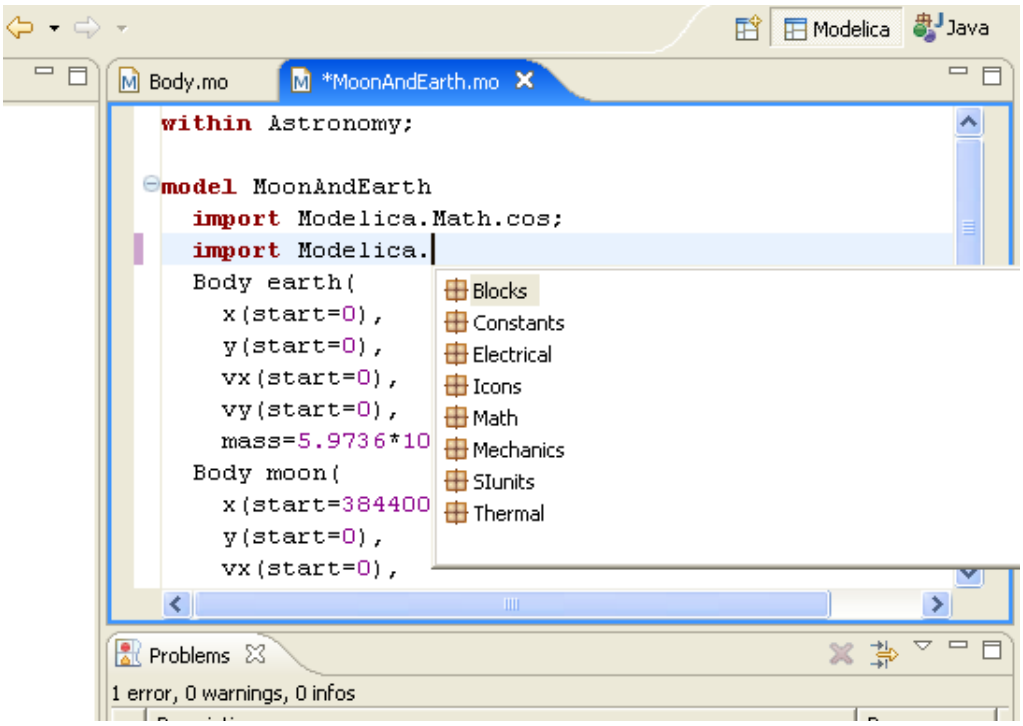

Figure 6: Code completion

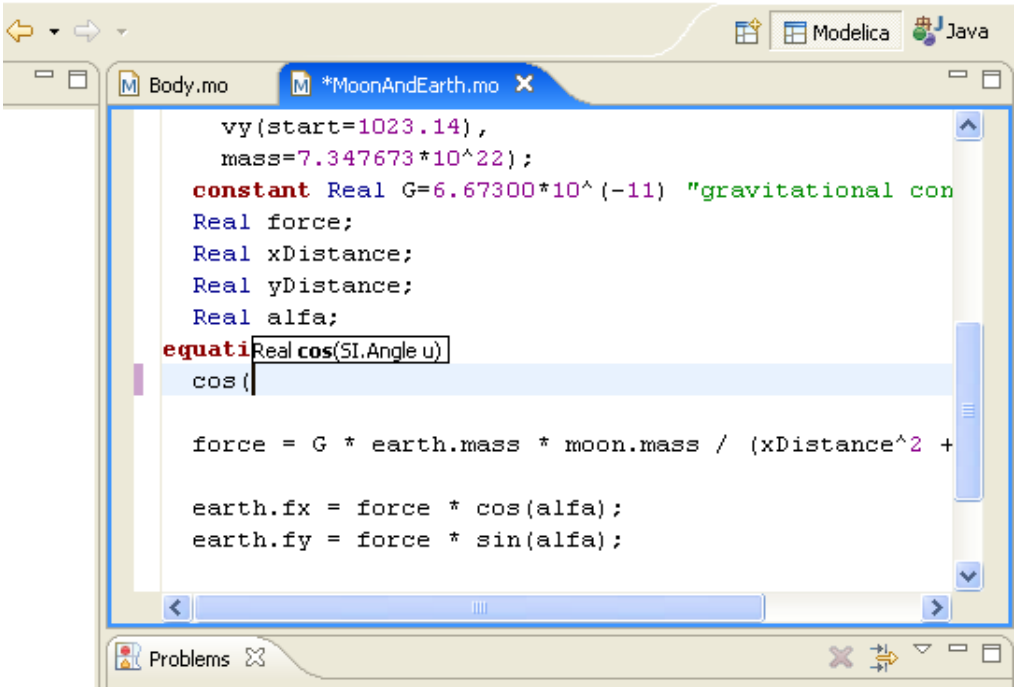

Figure 7: Information popup

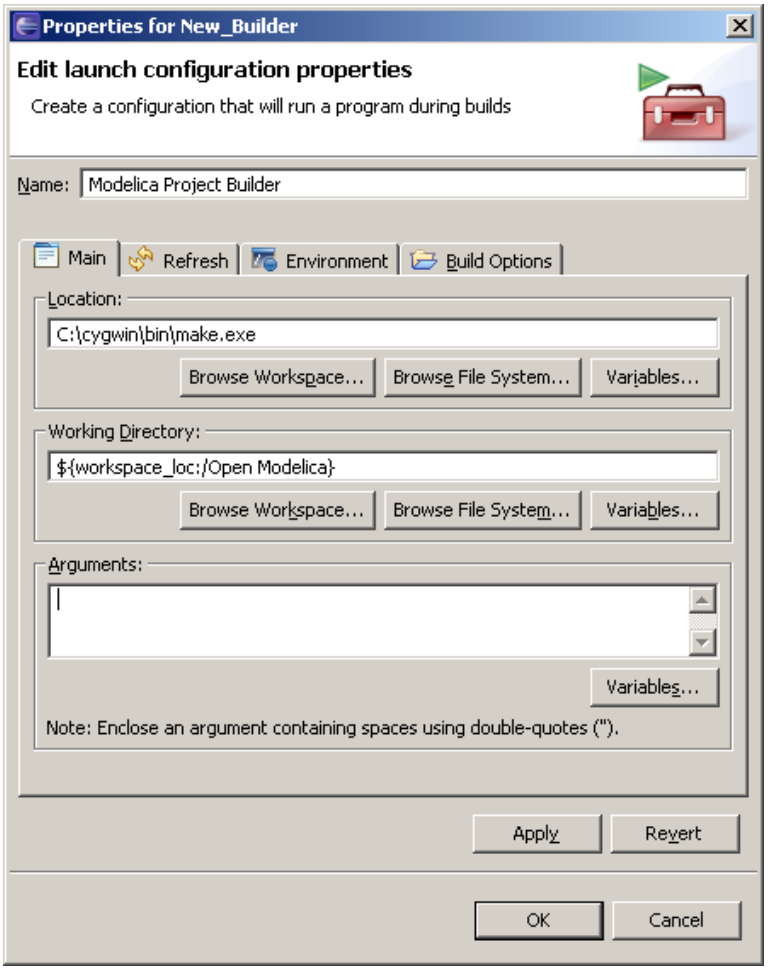

Figure 8: Properties for Modelica Project Builder

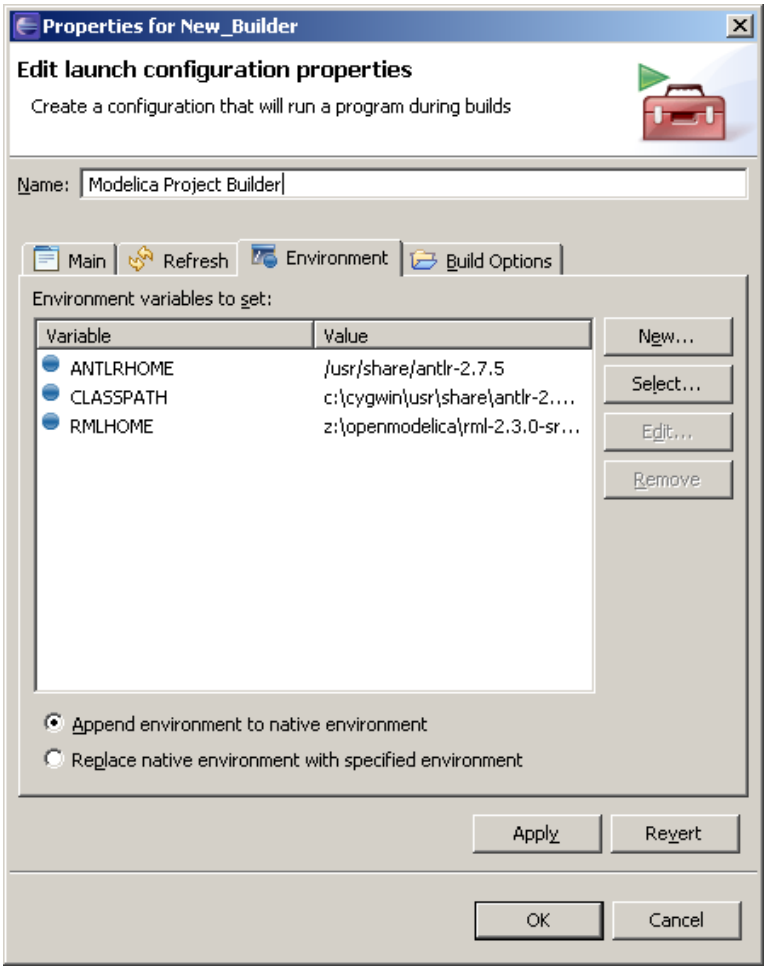

Figure 9: Environment properties

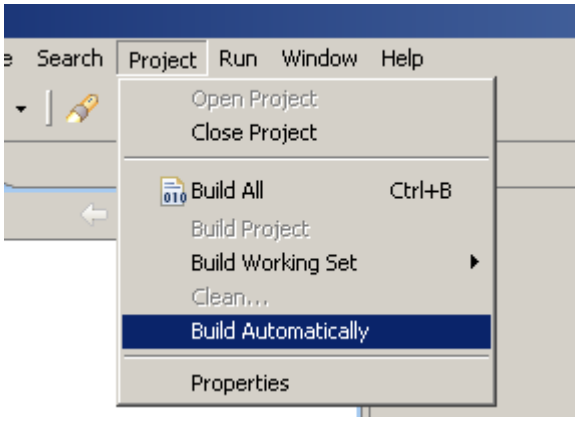

Figure 10: Deselect automatic building

4 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 core Modelica functionality, user interface and OpenModelica Compiler access services respectively. Together these plugins adds Modelica-specific functionality to the Eclipse IDE and creates an environment for working on Modelica projects.

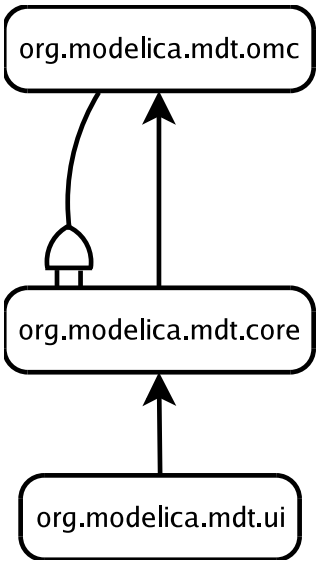

Figure 11: MDT Plugins Architecture

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.

4.1 org.modelica.mdt.core

The `core` plugin provides main functionality for MDT. It provides services needed by the `ui` plugin to implement the Modelica specific user interface. These services are Modelica elements hierarchies browsing, querying a mapping to the source code of the element, reversed querying of elements at a particular location in the source code, and a mechanism to track changes to elements.

4.1.1 Modelica Elements Access Layer

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

The elements 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.

4.1.2 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. User created Modelica and other types of resources are contained in the root folder of their respective project.

A special case of Modelica resources are classes defined in the standard library. These classes are accessible from the *IStandardLibrary* interface. See section 4.1.12 for more information.

4.1.3 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 to map a Modelica package to a folder structure, see section 10.3.3.2 in Peter Fritzson's book[26]. Any such folders are so called folder packages and are treated as packages by *IModelicaFolder*.

Modelica source files are represented by the *IModelicaSourceFile* interface. Modelica packages are represented by implementations of the *IModelicaClass* class. 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 *package*.

4.1.4 Source Files

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

caFile.getChildren().

4.1.5 Plain Files

All the files that does not have the extension *mo* are treated as plain files. Such files are represented by a wrapper interface *IModelicaFile*. This interface does not provide any additional services over the regular *IFile* interface, however it became apperent that it was convinient to be able to treat a plain file as a Modelica element.

4.1.6 Classes

The Modelica language defines 7 restrictions of classes. These restrictions are *model*, *function*, *record*, *connector*, *block*, *type* and *package*. The restriction type defines what restrictions are imposed on the class structure. There is also a special restriction called *class* which means that there are no restrictions on the contents of the class. See section 3.11.1 in Peter Fritzson's book[26] for more information on class restrictions.

Classes of all restrictions are represented by the *IModelicaClass* interface. The method *getRestriction()* can be used to query for the restriction of the class. A class can contain a myriad of Modelica elements. However at this point only a subset of possible elements are accessible. Subclasses and components are made accessible via the *getChildren()* method. The imports statments are returned by *getImports()*. If the class defines a function, the method *getSignature()* returns the input and output arguments.

4.1.7 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 section 3.11.1 in Peter Fritzson's book[26] 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.

4.1.8 Imports

The Modelica language defines three types of import statements: qualified, unqualified and renaming. All imports make available a new package in the

current class under a shorter name. The renaming import also provide the possibility to give the imported package a new name in the class where it is used, see a suitable resource for more information on import statements in Modelica.

Import statments are represented by the *IModelicaImport* interface. The type of import is available with the *getType()* method. The imported package is available via the *getImportedPackage()*. For the renaming imports, the new name of the package is available with *getAlias()*.

4.1.9 Function Signatures

Function signatures are represented by objects implementing the *ISignature* interface. The signature is basically a tuple of input and output parameters. Input parameters are queried with *getInputs()* and output parameters with *getOutput()*. Both methods returns arrays of *IParameter* objects.

The *IParameter* interface simply provides access to textual representations of a parameter's name and type. Use the self-documenting methods *getName()* and *getType()*, which both returns strings.

4.1.10 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 standard 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.

4.1.11 Reversed Mapping from the Source Code

The interface *IModelicaSourceFile* provides the *getClassAt()* method to query class definition at some position in the source file. This reversed mapping feature is used for example to provide code completions and infopops in the Modelica source code editor.

4.1.12 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 an instance of the *IStandardLibrary* interface. Such an object provides methods to browse and search for classes defined in the standard library. The exact contents of the standard library is determined by the currently used compiler plugin.

4.1.13 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.

4.1.14 Compiler Extension Point

The `core` plugin defines the 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 extension 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 an interface to a 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.

4.2 org.modelica.mdt.omc

The `omc` plugin provides access to the OpenModelica Compiler (OMC) for the `core` plugin. It does that by extending 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.

4.2.1 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, *sendExpression()*, that takes a String as argument and returns a String containing the OMC reply. The first time that an expression is going to be sent, communication with OMC will be established. If OMC can't be found when trying to contact it, it will have to be started.

There exists two interfaces that can be utilized to access OMC, one based on CORBA and one based on TCP sockets. We choose to utilize the CORBA interface as it gave us a higher level of abstraction, and the other components of the OpenModelica Environment uses the CORBA interface.

4.2.2 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 one of two possible ways to find the OMC executable. It either looks at the OPENMODELICAHOME environment variable, or it looks at the preference setting found by accessing *PreferenceManager.getCustomOmcPath()*.

The method used to start omc is determinded by the setting returned by the *PreferenceManager.getUseStandardOmcPath()* method, where the value provided by the OENMODELICAHOME environment variable is considered the standard path. The *PreferenceManager* class is defined in the *org.modelica.mdt.core.preferences* package.

4.2.3 OMC Interactive API

The interactive OMC API[16] 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.

4.2.4 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. An object is either a string or a list. The object lists and the error 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 *List* that can contain both *List*s and *ListElement*s. A *List* is basically a wrapper around a standard linked list. 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 generate an error. The error parser method is called *parseErrorString()* and can be found in the *OMCParser* class in the `omc` plugin.

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

The method *loadSourceFile()* takes a file name as argument and tries to load that file into OMC. This method will return a list of classes and packages found in the file along with any errors that are reported when loading the file.

The method *getClassLocation()* 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 *getRestriction()*.

The method *getElements()* 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 position information can be retrieved. Alot more information is available from this API function call, but are not used by MDT. See the documentation contained in the OMC source tree for a longer explanation of this and other API function calls.

4.3 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.

4.3.1 Modelica Development User Interface

Most of the functionality provided by the `ui` plugin is grouped in the Modelica perspective, see figure 12.

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

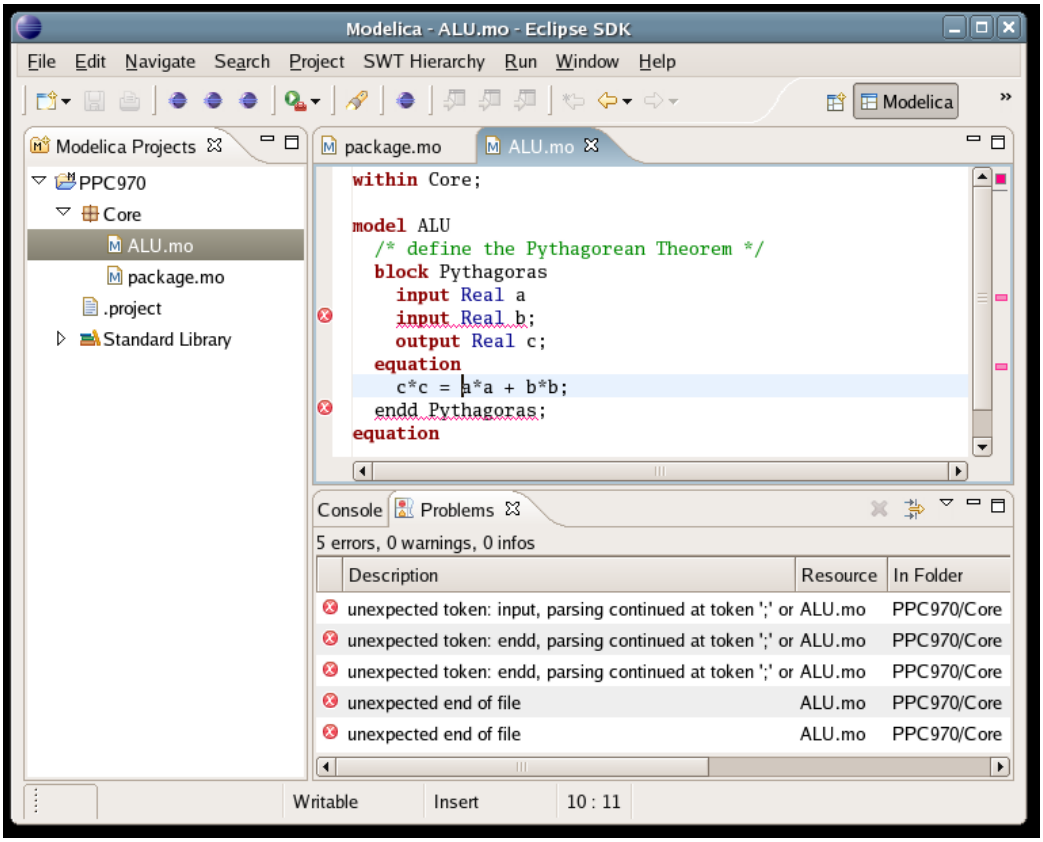

Figure 12: Modelica Perspective

4.3.2 Modelica Projects Browser

The `ui` plugin contributes a Modelica projects view. The view is largely inspired by the Java package browser. The Modelica projects view 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 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 *IModelicaRoot* 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 chapter 31 in Contributing to Eclipse[28] for more information. The `ui` plugin makes it possible to convert Modelica elements by installing an instance of *ModelicaElement-AdapterFactory* as an adapter factory. This is done in the *Plugin.start()* method. When the *WorkbenchLabelProvider* tries to convert a Modelica element to an *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.

4.3.3 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.

The method *openInEditor()* handles the details of opening an element in the 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 *ProjectsView-*

DoubleClickAction. 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!

4.3.4 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 the Modelica projects view.

4.3.5 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 the *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.

4.3.6 Abstract New Type Page

To create a new Modelica type, either a package or a class, the user must enter information on where this element should be created. Currently the user can specify the folder or the parent package. The user only need to specify either the folder or package, as the system is able to infer the other piece of the information automatically. The mapping between source folder and parent package is one to one, as currently creating packages inside the classes is not supported.

The above functionality is needed both in the new package and the new class wizards. To avoid duplication of the code, this functionality is implemented in an abstract wizard page defined in the *NewTypePage* class. This page defines two fields, *Source folder* and *Package*, and implements all the logic needed for these fields. It also implements some convenience functions

such as prefilling of the values based on the current selection, choosing folders and packages from special browse dialogs and other neatness.

The *NewTypePage* is not intended for direct use and therefore marked as an abstract class. The page is extended by the new package and new class wizards by adding specific fields suitable for package and class creation.

4.3.7 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 wizard. The page contains widgets for entering information on the new, soon to be created, Modelica package.

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.

4.3.8 New Class Wizard

The wizard is implemented by the *NewClassWizard* class. Figure 4 displays the new class wizard. The inner class *NewClassPage* implements the wizards only page. The wizard allows the user to select the restriction type of the class to be created. Based on the restriction type the wizard enables a set of 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.

The logic for enabling the modifiers checkboxes is implemented in the *NewClassPage.restrictionChanged()*. The method is invoked from an anonymous listener on the restriction widget.

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.

4.4 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, some 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 chapter 20 in Contributing to Eclipse[28] 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 chapter 20 in Contributing to Eclipse[28]. 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.internal.core.ErrorManager* class.

4.4.1 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.

4.4.2 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 *showCompil-*

erError() 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 the error notification policy.

4.4.3 Error Notification Policy

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, the 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.

4.4.4 Compiler Exceptions

org.modelica.mdt.core.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.

- *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.

4.5 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.

5 Regression Testing of MDT

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 could 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 [23].

5.1 Testing Tools

For running the regression tests on MDT, two special sets of Eclipse plugins are required, the JUnit PDE plugin and the Abbot 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 [9] for more information on creating and running tests with the JUnit framework.

Abbot 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 plugin must be installed separately, see [11] for details on obtaining and installing it.

See section 7.2 for a discussion about GUI testing and the tools used in the process.

5.2 Tests Plugin Project

The regression tests for MDT are all placed and run as the 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 tests 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.

5.3 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 to 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.

5.4 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 cases, 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.

5.5 Untested Code

Unfortunately writing regression tests for MDT often was given low priority 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 not tested by the regression tests and a number of bugs undiscovered. See the discussion on GUI testing in section 7.2 for more information. Also the fact that for one line of regression tests code there exists two lines of the production code suggests that there are large areas of untested code. Some of the known white spots of the tests 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.

5.6 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.

6 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.

6.1 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 non-Modelica projects from the Modelica Projects View.

6.2 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.

6.3 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.

6.4 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 believe 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 than it does now. See section 7.1.2 for a detailed discussion on the type of information that is required.

6.5 Quickfixes

Quickfixes should 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.

6.6 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 farfetched feature as currently there is no pressing need to use any other compiler besides OMC with MDT.

6.7 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[2] is an example of how a Modelica simulation environment with a graphical UI 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.

6.8 Testing

6.8.1 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 large parts of the Modelica Editor code. To get a complete picture of what tests are missing, testing coverage data should be generated.

6.8.2 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[3] should be investigated for the presence of a GUI recorder.

6.9 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*.

6.10 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.

7 Discussion and Related work

7.1 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 the projects view is dependent on the ability to parse the file.

In the future other OMC features should 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.

7.1.1 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 exist 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 unmarshals 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 marshaled, sent over the wire protocol, unmarshaled 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. Also, the extra layer consumes additional computing resources such as processor time and volatile storage.

A TCP socket protocol does not define a marshalling layer and thus necessitates 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 the wonders of automatic marshalling can be enjoyed. 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 were using the deprecated protocol. This must be taken into consideration before removing support.

We think that dropping the socket interface and moving marshalling features from the custom defined layer to the CORBA layer is the best approach.

7.1.2 Level of Information on Parsing

The amount of information OMC 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 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 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 6 on which possible feature would require what type of extra information from OMC.

Also, error reporting from the parsing phase should become 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. Hopefully this section will be obsolete by the time you read this, but don't hold your breath.

7.1.3 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 would 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 the Eclipse and OMC packages. Both OMC and MDT are uploaded to a yum repository. Eclipse 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 proprietary 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 world's packaging problems, this is someone else's work.

7.2 Testing of GUI Code

Writing regression tests for the GUI code turned out to be quite a bit more problematic than 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[3].

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 how 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 SWT threading issues[20] 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, the writing of 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. There probably 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 a 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.

7.3 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 singled 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. If communication were to be performed by utilizing a more CORBA-oriented interface, "unexpected reply" would never occur.

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.

7.4 The Modelica Package Structure

The Modelica package structure is quite different to other programming languages packages structures, and we will compare it with how Java does in this section.

In Java, the package structure is directly mapped to the file system. This means that a class that is in the package `org.modelica.mdt` will be found in the folder org/modelica/mdt. This one-to-one mapping makes it easy to find classes and easy for an environment to browse classes found in a package.

In Modelica, a package is just a restriction of a normal class. This means that a package can be a part of a class. But this is not the only way that a package can be defined, it can also be defined as a directory in the filesystem that contains a file named package.mo. This means that there are several different ways that entities on disk will be represented as packages in Modelica and in MDT.

If Modelica dropped support for having packages inside of classes and only supported packages that were directories, the Modelica package structure

would more resemble the Java package structure.

Having packages as part of classes is the wrong way around, classes should be placed inside of packages. As a package is just a name-space divider, it should not be a part of the class-restriction hierarchy.

7.5 Other Modelica Development Environments

There are many other environments for developing Modelica projects. Some of the environments are proprietary and commercial, while others are only proprietary. Some are free software[6].

The following descriptions of other environments also point out the differences between the environments and MDT. Many of the following environments are just editors with only some of the features that we feel are needed for a good Modelica environment. Another important aspect of software is its portability, and a couple of the environments are only available for Microsoft Windows.

7.5.1 Dymola

Dymola is developed and maintained by the company Dynasim[2]. It is a complete modeling and simulation environment which also has support for graphical modeling. As Dymola is commercial and proprietary software, MDT is not exactly in its league.

The two essential things that are missing from MDT to make it somewhat equal to Dymola are simulation and graphical editing of models. These features are probably also the most useful features of a modeling environment.

To get an idea of how Dymola looks like, see Figure 13 on page 47.

7.5.2 MathModelica

MathModelica is a Modelica environment developed by MathCore[10]. It is, as Dymola, a complete modeling and simulation environment. The old version of MathModelica uses the Dymola kernel to actually do the simulations, whereas the new version will use OpenModelica with some extensions. The current graphical editor used in MathModelica depends on Microsoft Visio. Another graphical editor has been developed that is not dependent on proprietary software. See Figure 14 on page 48 to see how this new graphical editor looks like.

As MathModelica currently depends on Microsoft Visio, it is not portable to anything else than various versions of Windows.

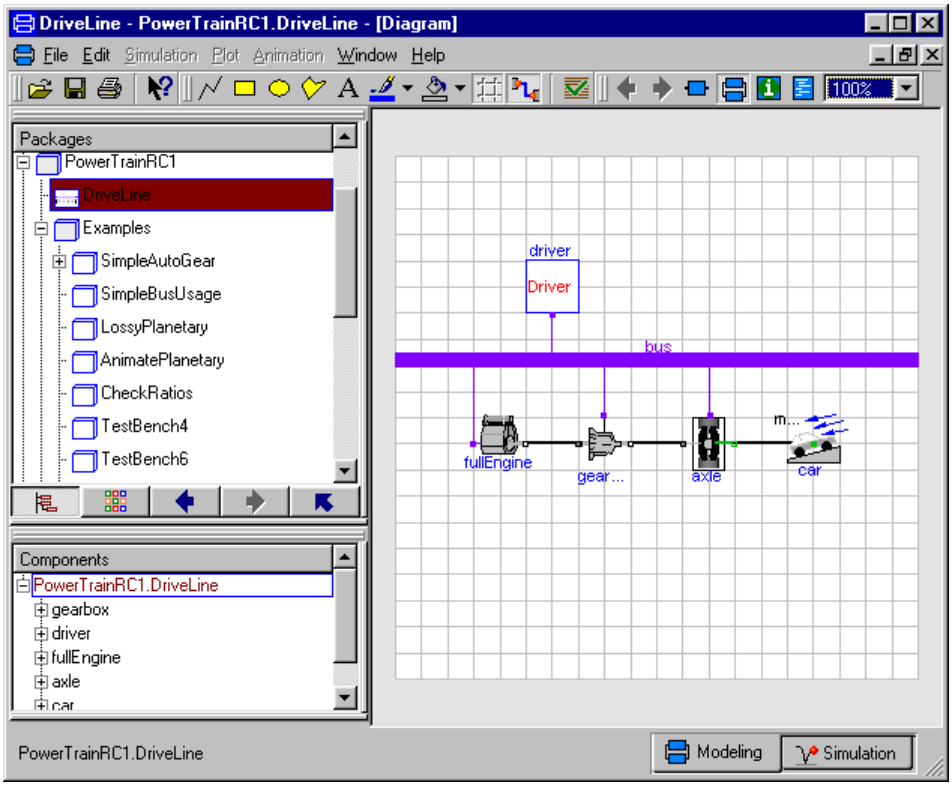

Figure 13: Dymola

7.5.3 Free Modelica Editor

The Free Modelica Editor[5] (FME) was created by Falko Jens Wagner at the Technical University of Denmark. Among the features found are syntax highlighting, object browsing, and code templates. See Figure 15 on page 49 for an example of how a session with FME can look like. FME is only available for Microsoft Windows.

In comparison with MDT, the Free Modelica Editor only works on source files and not whole projects. This means for example that the object browser only displays Modelica classes and packages that are found in the current file that is being edited. When another file is selected, only that files contents will be displayed. This is a big drawback when working on larger projects with several files.

There are some things that FME can do that MDT can't. FME can do simulations of Modelica files by sending the file and different parameters to Dymola[2]. That this feature is missing from MDT is a big downside, as simulation of models is a big part of Modelica development. As a side note, you can actually do simulations from MDT but it requieras a hack. You can

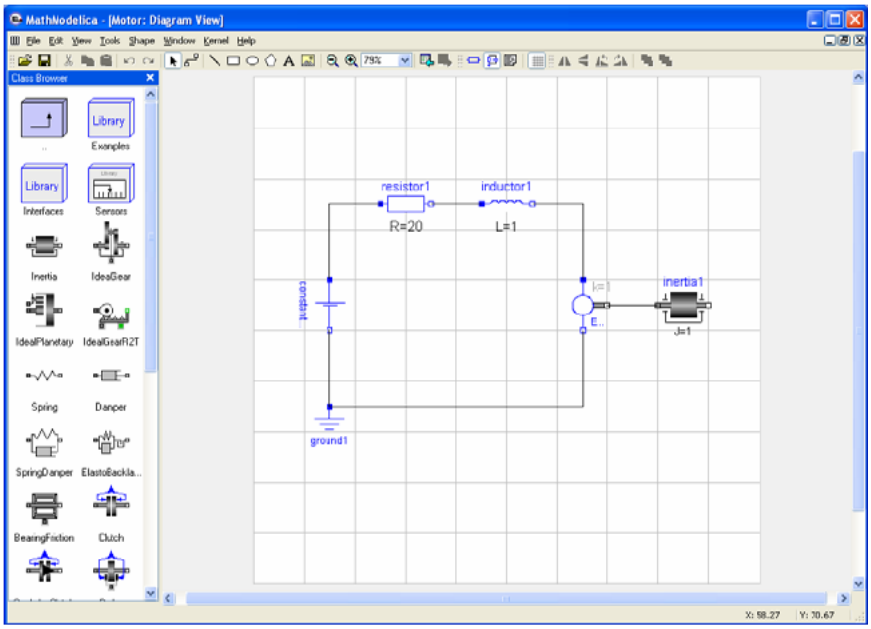

Figure 14: MathModelica

create an Eclipse external program builder that runs any program you like, and thereby run a simulation. This is not at all as helpful as actually having simulations available from the interface.

Another thing that FME can do is hiding of annotations. A problem is that this feature is implemented by rewriting the file, inserting a placeholder for the annotation with a short code. An annotation can for example be replaced by the string "//ann.01". If you by mistake remove some part of this coded replacement, the annotations that are held in memory can be destroyed.

A final problem with FME is that it is quite old and outdated (last updated in 2000). MDT will hopefully be kept up-to-date with new developments in the Modelica community.

7.5.4 Modelica Mode for GNU Emacs

GNU Emacs[7] is an extensible editor originally created by Richard Stallman[8]. As it is a free software project, many other developers have contributed throughout the years. It can easily be extended by using a variant of Lisp.

A Modelica mode for Emacs has been developed by Ruediger Franke. This mode has support for syntax highlighting, annotation hiding and object browsing by using the OO-Browser[15]. See Figure 16 on page 50. The

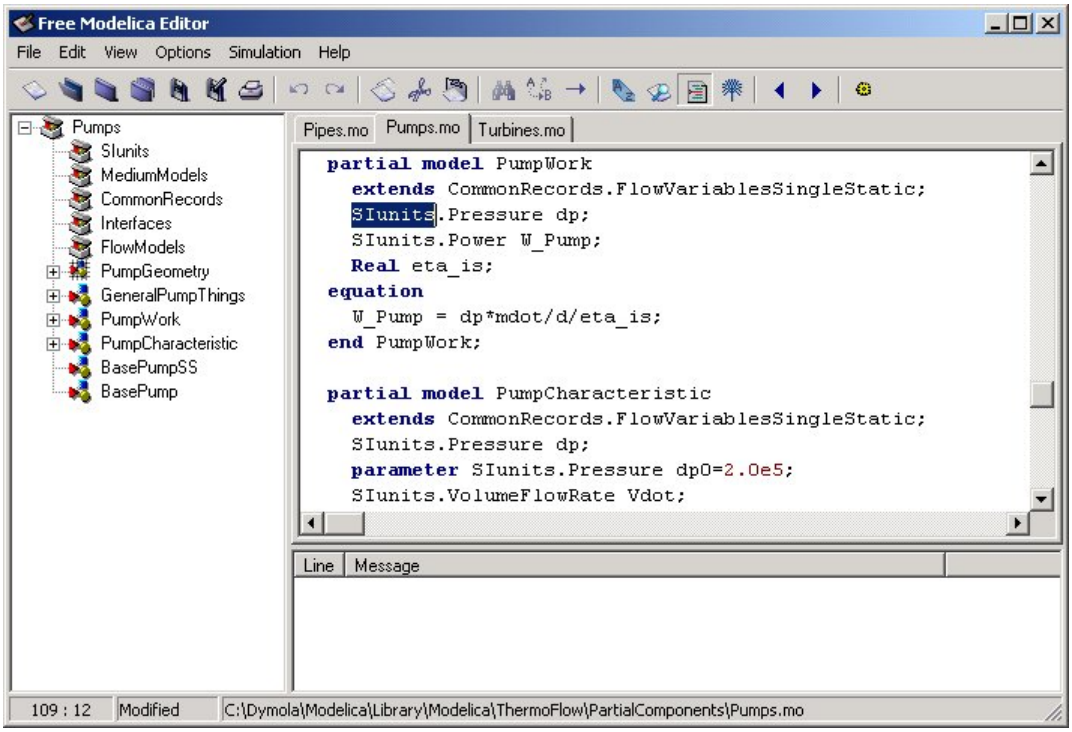

Figure 15: The Free Modelica Editor

Modelica mode can be found on the OpenModelica website[17].

Installing the Modelica mode is alot of hassle, you have to update some files and also change some of the code in the OO-Browser. Another downside is that you have to be rather familiar with Emacs to be able to use this mode.

As GNU Emacs runs on most modern operating systems, this environment is probably portable.

7.5.5 SciTE with Modelica Mode

SciTE is the Scintilla Text Editor[12] and has been developed by the MOSILAB[14] group. Scintilla can be seen as a framework for creating text editors for computer languages, and SciTE is an editor that is implemented by using this framework. As Scintilla is aimed at both the Win32 API and GTK+, it's widely portable and should be able run on most modern operating systems. See Figure 17 on page 51.

A mode for editing Modelica is available, and it has a couple of features. As can be expected, syntax highlighting is available. There is also support for code folding, including annotations, and some support for code completion in the form of Modelica keywords and identifiers.

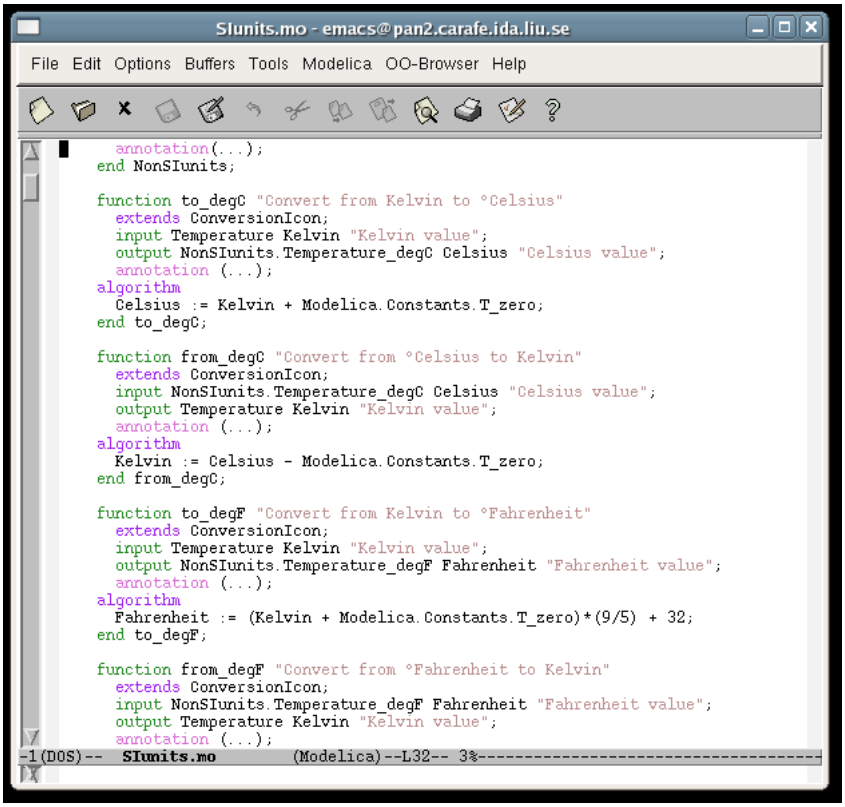

Figure 16: Modelica mode for GNU Emacs

There's no syntax checking, and no project browser is made available. These two features are missing from most editors as this requires access to a Modelica parser. SciTE probably has a parser to be able to perform code folding, and it could probably have been used for some kind of error notification.

7.5.6 UltraEdit with Modelica Keywords

UltraEdit[21] is an advanced editor for programmers. It handles text, HTML, PHP, Java, Perl and Javascript. A Modelica "mode" for UltraEdit can be found at the Modelica Association website[13]. This mode only contains syntax highlighting of Modelica keywords, comments and Modelica identifiers.

To get the Modelica features to work, you have to manually copy text from the file found at the Modelica Association to a specific file in the UltraEdit installation. We find this installation method quite awkward.

As UltraEdit is only an editor, it has no real idea of how Modelica code can and should look like. There's no project browser and no syntax checking,

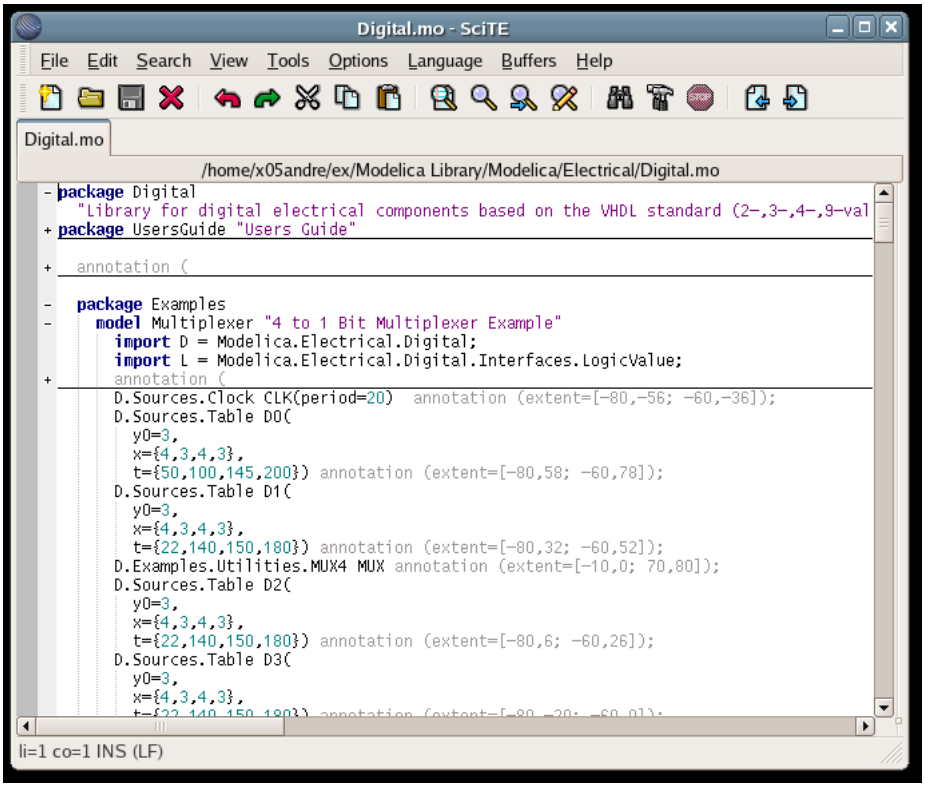

Figure 17: SciTE

making it a lot less useful than other Modelica environments.

UltraEdit is only available for Microsoft Windows.

8 Conclusions

8.1 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 and information popups.

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 7.1.1.

8.2 What We Deliver

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

8.2.1 The Plugins

The three plugins (`core`, `ui`, `omc`) that we provide can be easily fetched by using the provided update site[22]. You can visit the update site with a web browser to get instructions on how to use the update site from Eclipse.

8.2.2 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 Users Guide.

The development documents are located in the MDT directory of the OpenModelica subversion repository. The OpenModelica website[18] provides details about reaching this repository.

8.2.3 Source Code

The source code of MDT is available in the OpenModelica subversion repository. See the OpenModelica website[18] for details about accessing the repository. Each plugin (`core`, `ui`, `omc`) is available as separate Eclipse projects. A tip is to use Subclipse[19] from within Eclipse to access the subversion repository.

A Package Overview

This is an overview of the Java packages that compose the MDT plugins and regression tests. A short description of each package is provided. For full information about packages and their contents, please see the source code.

org.modelica.mdt.core Plugin

org.modelica.mdt.core

This package defines the public interface to the services provided by the `core` plugin. All clients should access the core functionality only through this interface.

org.modelica.mdt.core.builder

This package contains the implementation of the Modelica syntax checker which is configured to run as an incremental builder.

org.modelica.mdt.core.compiler

This package defines the Modelica compiler extension point access interface. Plugins that wish to contribute access to a Modelica compiler should use the classes in this package.

org.modelica.mdt.core.preferences

This package contains the preference manager which allows reading and writing user preferences.

org.modelica.mdt.internal.core

Here is the implementation of the public interface defined in the *org.modelica.mdt.core* package.

org.modelica.mdt.omc Plugin

org.modelica.mdt.omc

This package implements the Modelica compiler extension point to provide access to the OpenModelica Compiler.

org.modelica.mdt.omc.internal

This package contains implementation details of the org.modelica.mdt.omc package, which are not suitable to expose to clients.

org.modelica.mdt.omc.internal.corba

This package contains the automatically generated code from the CORBA IDL definition file. This definition file is part of the OpenModelica Compiler source code distribution. In the unlikely event of changes to the CORBA interface of the OpenModelica Compiler, the classes in this package should be replaced with newly generated files from the definition file.

org.modelica.mdt.ui Plugin

org.modelica.mdt.ui

This package contains generic code that is used by other packages in this plugin.

org.modelica.mdt.ui.editor

The Modelica source code editor is implemented in this package. Code for code completion, context information, syntax highlighting, and support for opening Modelica elements from the Modelica Project view.

org.modelica.mdt.ui.preferences

This package contributes the Modelica Preference page.

org.modelica.mdt.ui.view

This package contains the Modelica Project view that allows package browsing.

org.modelica.mdt.ui.wizards

This package contains wizards for creation of Modelica projects, packages, and classes.

org.modelica.mdt.test Plugin

org.modelica.mdt.test

This package contains the regression tests for MDT. They are implemented as JUnit test cases.

org.modelica.mdt.test.util

This package contains some helper code for running and writing testcases.

References

[1] A brief history of eclipse. `http://www-128.ibm.com/developerworks/rational/library/nov05/cernosek/`.

[2] Dynasim. `http://www.dynasim.se`.

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

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

[5] Free modelica editor. `http://www.et.web.mek.dtu.dk/FME/index.html`.

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

[7] Gnu emacs. `http://www.gnu.org/software/emacs/`.

[8] Homepage of richard stallman. `http://www.stallman.org/`.

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

[10] Mathcore. `http://www.mathcore.com`.

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

[12] Modelica editor. SciTE `http://www.mosilab.de/downloads/software/modelica-editor-scite`.

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

[14] Mosilab at fraunhofer-berlin. `http://www.mosilab.de/`.

[15] The oo-browser. `http://sourceforge.net/projects/oo-browser/`.

[16] Openmodelica system documentation. `http://www.ida.liu.se/labs/pelab/modelica/OpenModelica/Documents/OpenModelicaSystem-050830.pdf`.

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

[18] Openmodelica website. `http://www.ida.liu.se/labs/pelab/modelica/OpenModelica.html`.

[19] Subclipse. `http://subclipse.tigris.org/`.

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

[21] Ultraedit. `http://www.ultraedit.com/`.

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

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

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

[25] Susan A. Dart, Robert J. Ellison, Peter H. Feiler, and A. Nico Habermann. Software development environments. In E. J. Chikofsky, editor, *Computer-Aided Software Engineering (CASE)*. IEEE Computer Society Press, 1989.

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

[27] Peter Fritzson, Adrian Pop, and Peter Aronsson. Towards comprehensive meta-modeling and meta-programming capabilities in modelica. 2005. `http://www.modelica.org/events/Conference2005/`.

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