OpenModelica Development Environment with Eclipse Integration for Browsing, Modeling, and Debugging

Adrian Pop, Peter Fritzson, Andreas Remar, Elmir Jagudin, David Akhvlediani
PELAB – Programming Environment Lab, Dept. Computer Science
Linköping University, S-581 83 Linköping, Sweden
{adrpo, petfr}@ida.liu.se

Abstract

The OpenModelica (MDT) Eclipse Plugin integrates the OpenModelica compiler and debugger with the Eclipse Integrated Development Environment Framework.. MDT, together with the OpenModelica compiler and debugger, provides an environment for Modelica development projects. This includes browsing, code completion through menus or popups, automatic indentation even of syntactically incorrect models, and model debugging. Simulation and plotting is also possible from a special command window. To our knowledge, this is the first Eclipse plugin for an equation-based language.

1 Introduction

The goal of our work with the Eclipse framework integration in the OpenModelica modeling and development environment is to achieve a more comprehensive and more powerful environment. It can be useful to first take a general look at this area including some background.

1.1 Integrated Interactive Programming Environments

An integrated interactive modeling and simulation environment is a special case of programming environments with applications in modeling and simulation. Thus, it should fulfill the requirements both from general integrated environments and from the application area of modeling and simulation mentioned in the previous section.

The main idea of an integrated programming environment in general is that a number of programming support functions should be available within the same tool in a well-integrated way. This means that the functions should operate on the same data and program representations, exchange information when necessary, resulting in an environment that is both powerful and easy to use. An environment is interactive and incremental if it gives quick feedback, e.g. without recomputing everything from scratch, and maintains a dialogue with the user, including preserving the state of previous interactions with the user. Interactive environments are typically both more productive and more fun to use.

There are many things that one wants a programming environment to do for the programmer, particularly if it is interactive. What functionality should be included? Comprehensive software development environments are expected to provide support for the major development phases, such as:

- Requirements analysis.
- Design.
- Implementation.
- Maintenance.

A programming environment can be somewhat more restrictive and need not necessarily support early phases such as requirements analysis, but it is an advantage if such facilities are also included. The main point is to provide as much computer support as possible for different aspects of software development, to free the developer from mundane tasks, so that more time and effort can be spent on the essential issues. The following is a partial list of integrated programming environment facilities, some of which are already mentioned in (Sandewall 1978 [11]), that should be provided for the programmer:

- Administration and configuration management of program modules and classes, and different versions of these.
- Administration and maintenance of test examples and their correct results.
- Administration and maintenance of formal or informal documentation of program parts, and automatic generation of documentation from programs.
- Support for a given programming methodology, e.g. top-down or bottom-up. For example, if a top-down approach should be encouraged, it is natural for the interactive environment to maintain successive

composition steps and mutual references between those.

- Support for the interactive session. For example, previous interactions should be saved in an appropriate way so that the user can refer to previous commands or results, go back and edit those, and possibly re-execute.
- Enhanced editing support, performed by an editor that knows about the syntactic structure of the language. It is an advantage if the system allows editing of the program in different views. For example, editing of the overall system structure can be done in the graphical view, whereas editing of detailed properties can be done in the textual view.
- Cross-referencing and query facilities, to help the user understand interdependences between parts of large systems.
- Flexibility and extensibility, e.g. mechanisms to extend the syntax and semantics of the programming language representation and the functionality built into the environment.
- Accessible internal representation of programs. This is often a prerequisite to the extensibility requirement. An accessible internal representation means that there is a well-defined representation of programs that are represented in data structures of the programming language itself, so that user-written programs may inspect the structure and generate new programs. This property is also known as the principle of program-data equivalence.

Early work in interactive integrated programming environments supporting a specific language was done in the InterLisp system for the Lisp language: (Teitelman 1974 [12]), common principles and experience of early interactive Lisp environments are described in (Sandewall 1978 [11]), interactive and incremental Pascal with the DICE system: (Fritzson 1983 [3]), the integrated Mjölner environment, (Lindskov, Knudsen, Lehrmann-Madsen, and Magnusson 1993 [9]).

1.2 The Eclipse Framework

Eclipse [1] is an open source framework for creating extensible integrated development environments (IDEs). (For the history of Eclipse, see Section 6). One of the goals of the Eclipse platform is to avoid duplicating common code that is needed to implement a powerful integrated environment for development of software. By allowing third parties to easily extend the platform via the plugin concept, the amount of new code that needs to be written is decreased.

1.3 Eclipse Platform Architecture

By itself, Eclipse does not provide much end-user functionality. The important contributions to Eclipse are based on its 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 (JDT) is for example a collection of plugins that are loaded into Eclipse when they are requested. The fact 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 capabilities. 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.

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 the 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 is the file name extension that selects what editor 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 makes the memory impact reasonably low while running Eclipse.

A user-friendly aspect of Eclipse is the Eclipse Update Manager which allows the user to install new plugins just by pointing Eclipse to a certain website. This website is provided by the developers of the plugin that the user may wish to install. An update site at the OpenModelica [13] web site is for example provided for easy installation of the latest version of MDT.

1.4 OpenModelica MDT Plugin into Eclipse

The MDT Eclipse plugin provides file and class hierarchy browsing and text editing capabilities. Syntax highlighting facilities and a compilation manager are also included in MDT, as well as integration of the debugger for the algorithmic Modelica code.

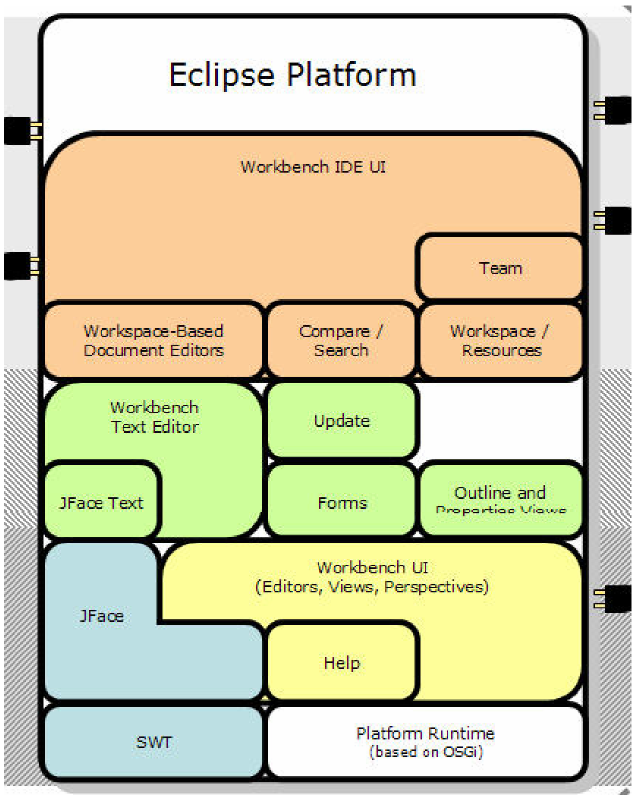

Eclipse Platform
Workbench IDE UI
Team
Workspace-Based Document Editors
Compare / Search
Workspace / Resources
Workbench Text Editor
Update
JFace Text
Forms
Outline and Properties Views
JFace
Workbench UI (Editors, Views, Perspectives)
Help
SWT
Platform Runtime (based on OSGi)

Figure 1. The architecture of Eclipse, with possible plugin positions marked.

The Eclipse framework (Figure 1) has the advantage of making it easy to add future extensions.

2 OpenModelica Environment Architecture

The MDT Eclipse plugin is integrated in the OpenModelica environment which consists of several interconnected subsystems, as depicted in Figure 2 below.

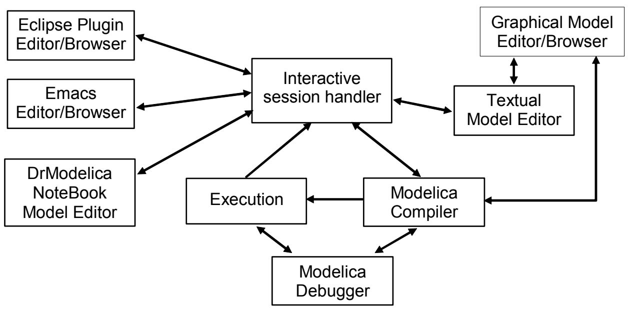

Eclipse Plugin Editor/Browser
Emacs Editor/Browser
DrModelica NoteBook Model Editor
Interactive session handler
Graphical Model Editor/Browser
Textual Model Editor
Execution
Modelica Compiler
Modelica Debugger

Figure 2. The architecture of the OpenModelica environment.

Arrows denote data and control flow. Several subsystems provide different forms of browsing and textual editing of the Modelica code.

OpenModelica is structured as several communicating processes in client-server architecture, primarily exchanging information through a Corba interface, see Figure 3. The OpenModelica compiler/interpreter (OMC) is the server, communicating with clients. The Eclipse MDT plugin is one of the clients.

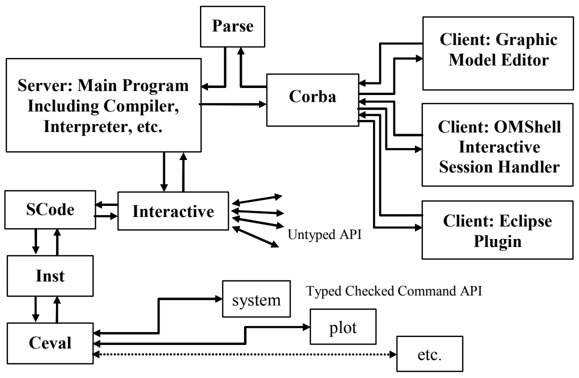

Parse
Server: Main Program Including Compiler, Interpreter, etc.
Corba
Client: Graphic Model Editor
Client: OMShell Interactive Session Handler
Client: Eclipse Plugin
SCode
Interactive
Untyped API
Inst
system
Typed Checked Command API
plot
Ceval
etc.

Figure 3. The client-server architecture of the OpenModelica environment.

Messages from the Corba interface are of two kinds. The first group consists of expressions or user commands which are evaluated by the Ceval module. The second group includes declarations of classes, variables, etc., assignments, and client-server API calls that are handled via the Interactive module, which also stores information about interactively declared/assigned items at the top-level in an environment structure

3 Modelica Development Tooling (MDT) Eclipse Plugin

As mentioned, the Modelica Development Tooling (MDT) Eclipse Plugin provides an environment for working with Modelica development projects.

The following features are available:

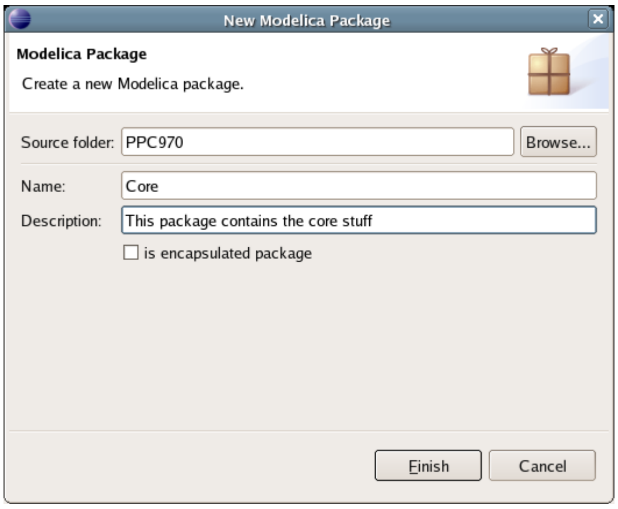
- Browsing support for Modelica projects, packages, and classes.
- Wizards for creating Modelica projects, packages, and classes.
- Syntax color highlighting.
- Syntax checking.
- Code completion when writing code to reference a class.
- Code completion/signature information when writing function calls.
- Browsing of the Modelica Standard Library and other Modelica package hierarchies.
- Support for MetaModelica extensions to standard Modelica.

3.1 Using the Modelica Perspective

The most convenient way to work with Modelica projects is to use to the Modelica perspective. To switch to the Modelica perspective, choose the `Window` menu item, select `Open Perspective` followed by `Other...` Select the `Modelica` option from the dialog presented and click `OK`.

3.2 Creating a Project

To start a new project, use the `New Modelica Project` Wizard. It is accessible through `File->New-> Modelica Project` or by right-clicking in the Modelica Projects view and selecting `New->Modelica Project`.



Modelica Package

Create a new Modelica package.

Source folder: PPC970

Name: Core

Description: This package contains the core stuff

☐ is encapsulated package

Figure 4. Creating a new package.

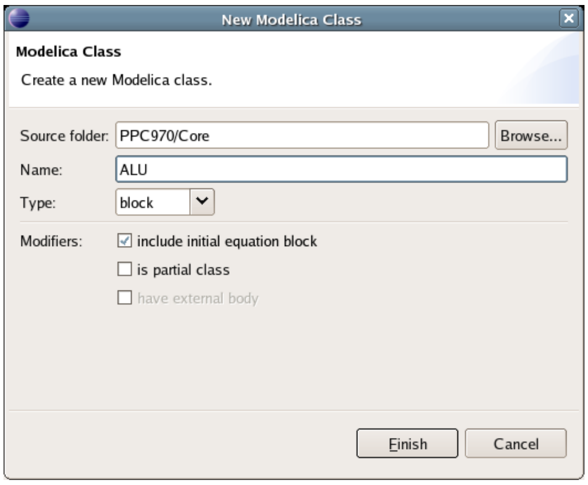
3.3 Creating a Package

To create a new package inside a Modelica project, select `File->New->Modelica Package`. Enter the desired name of the package and a description of what it contains.

3.4 Creating a Class

To create 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 type, you 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.



Modelica Class

Create a new Modelica class.

Source folder: PPC970/Core

Name: ALU

Type: block

Modifiers: ☑ include initial equation block

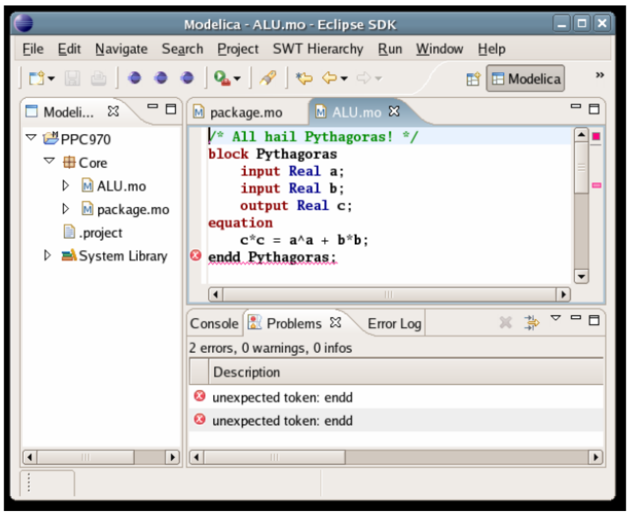
☐ is partial class

☐ have external body

Figure 5. Creating a new class.

3.5 Syntax Checking

Whenever a Modelica (`.mo`) file is saved by the Modelica Editor, it is checked for syntactical errors. Any errors found are added to the Problems view and also marked in the source code editor.



```
/* All hail Pythagoras! */
block Pythagoras
    input Real a;
    input Real b;
    output Real c;
equation
    c*c = a^a + b*b;
endd Pythagoras;
```

2 errors, 0 warnings, 0 infos

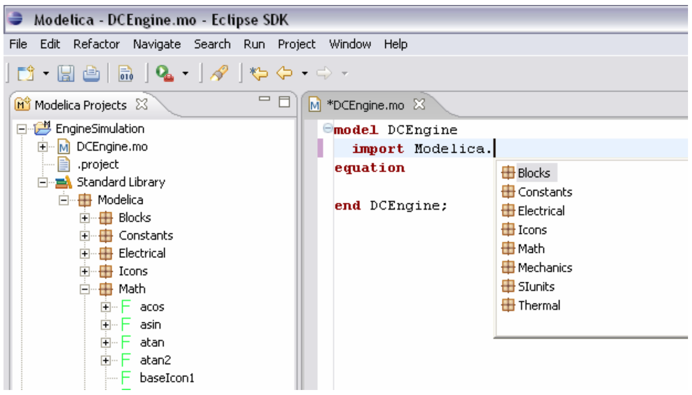
Description
unexpected token: endd
unexpected token: endd

Figure 6. Syntax checking.

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. To reach the problem, one can either click the item in the Problems view or select the red box in the right-hand side of the editor.

3.6 Code Completion

MDT supports Code Completion in two variants. The first variant, code completion when typing a dot after a class (package) name, shows alternatives in a menu:

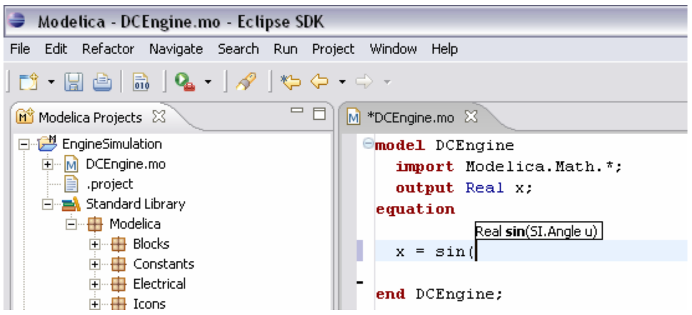


```
model DCEngine
  import Modelica.
equation

end DCEngine;
```

Figure 7. Code completion using a popup menu after a dot

The second variant is useful when typing a call to a function. It shows the function signature (formal parameter names and types) in a popup when typing the parenthesis after the function name, here the signature `Real sin(SI.Angle u)` of the `sin` function:



```
model DCEngine
  import Modelica.Math.*;
  output Real x;
equation

  x = sin(

end DCEngine;
```

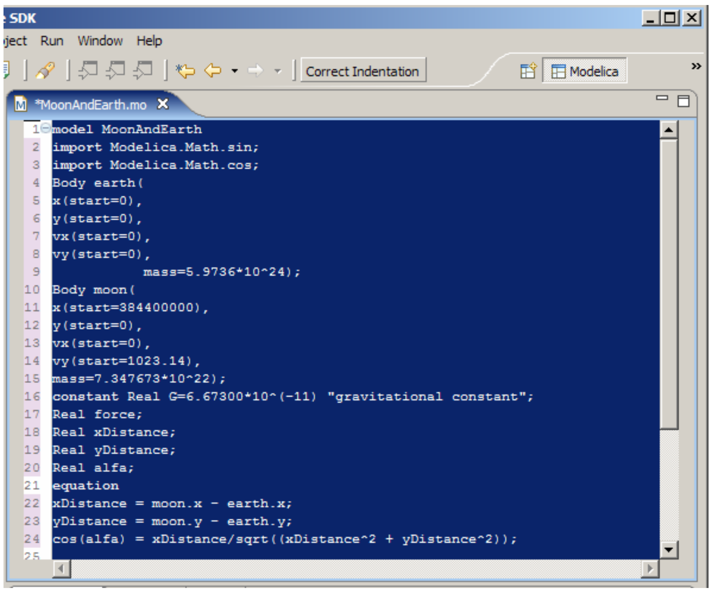
Figure 8. Code completion showing a popup function signature after typing a left parenthesis.

3.7 Automatic Indentation

MDT has recently obtained support for automatic indentation. When typing the Return (Enter) key, the next line is indented correctly. You can also correct indentation of the current line or a range selection using CTRL+I or "Correct Indentation" action on the toolbar or in the Edit menu.

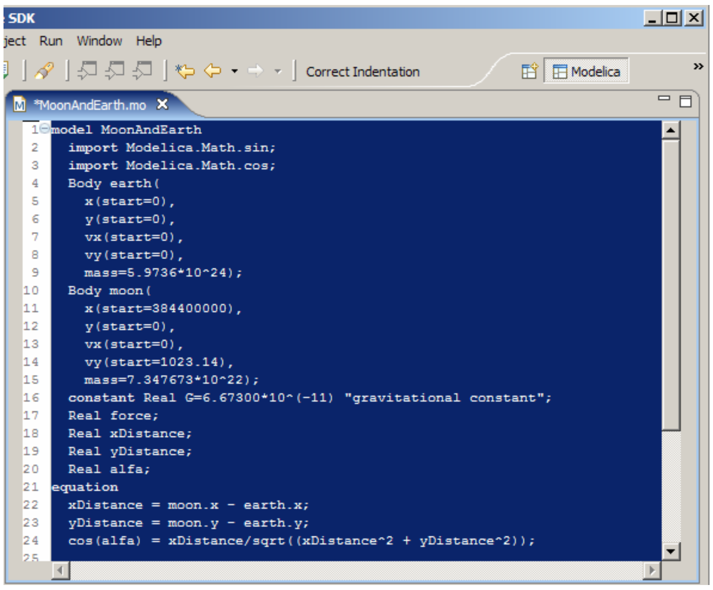
Indentation can be applied to incomplete code as a heuristic Modelica scanner is used and the indentation is based only on the tokens generated by this scanner. The indenter indents one line at a time. For example, consider that line four (4) in Figure 10 should be indented. The indenter asks the heuristic scanner to give tokens from the starting token in backwards direction to the start of the file until a scope introducer is recognized, which for this particular file is `model MoonAndEarth`. The reference position of the start of the scope introducer is computed and line four (4) is indented from this reference position one indent unit. The indentation result is presented in Figure 10.

Indenting Modelica code is far from trivial when incomplete (possibly incorrect) code should be indented correctly. Most of the difficulty comes from Modelica scopes which are hard to recognize using just a scanner and some logic behind it. In languages like C/C++ and Java finding enclosing scopes is very easy as one character tokens are used for the scope opening and closing: `"{"` and `"}"`. In Modelica you need at least two tokens and much more case analysis to find where a scope starts and ends. Complications also arise when mixing if-statements with if-expressions (which was further complicated by the introduction of conditional declarations in the Modelica language). In this particular case we implemented a parser emulator that recognizes these constructs based on scanner tokens delivered backwards.



```
model MoonAndEarth
import Modelica.Math.sin;
import Modelica.Math.cos;
Body earth(
x(start=0),
y(start=0),
vx(start=0),
vy(start=0),
            mass=5.9736*10^24);
Body moon(
x(start=384400000),
y(start=0),
vx(start=0),
vy(start=1023.14),
mass=7.347673*10^22);
constant Real G=6.67300*10^(-11) "gravitational constant";
Real force;
Real xDistance;
Real yDistance;
Real alfa;
equation
xDistance = moon.x - earth.x;
yDistance = moon.y - earth.y;
cos(alfa) = xDistance/sqrt((xDistance^2 + yDistance^2));
```

Figure 9. Example of code before indentation.



```
1  model MoonAndEarth
2    import Modelica.Math.sin;
3    import Modelica.Math.cos;
4    Body earth(
5      x(start=0),
6      y(start=0),
7      vx(start=0),
8      vy(start=0),
9      mass=5.9736*10^24);
10   Body moon(
11     x(start=384400000),
12     y(start=0),
13     vx(start=0),
14     vy(start=1023.14),
15     mass=7.347673*10^22);
16   constant Real G=6.67300*10^(-11) "gravitational constant";
17   Real force;
18   Real xDistance;
19   Real yDistance;
20   Real alfa;
21 equation
22   xDistance = moon.x - earth.x;
23   yDistance = moon.y - earth.y;
24   cos(alfa) = xDistance/sqrt((xDistance^2 + yDistance^2));
```

Figure 10. Example of code after automatic indentation.

The indenter works in almost all cases, but there are cases in which is impossible to find the correct indentation. For example when the indentation of a line consisting of `"end Name;"` is requested and the scope introducer for `Name` is not found (that is identifier `Name` followed backwards by `class`, `model`, `package`, `block`, `record`, `connector` etc.) then the indenter fails and returns the indentation of the previous line.

4 The OpenModelica Debugger Integrated in Eclipse

We have integrated our debugger for algorithmic Modelica code (Adrian Pop and Peter Fritzson, 2005 [10]) within the Eclipse debugging framework.

The communication protocol between MDT and the debugger (which is included in the compiled executable built for simulation) is based on a client-server architecture and is implemented via sockets. The debugger is the server and MDT is the client.

When the debugged model is simulated, the debugger receives from MDT all the breakpoints set within the algorithmic code. Then the debugger resumes the application program. When a breakpoint condition becomes true the debugger stops the program and listens to commands it may receive from MDT. The commands accepted by the MDT client are classic: variable value printing, stack trace printing, stepping, running, etc. MDT sends appropriate commands to the debugger, parses the information received and displays it within the MDT debugging views to be inspected by the programmer.

Because algorithmic code can be executed millions of times within a simulation, it is very important to be able to specify breakpoints based on variable values and/or the number of times a function executes. These types of breakpoints were recently added to the debugging framework previously described in (Adrian Pop and Peter Fritzson, 2005 [10]) and are now available, also in MDT within Eclipse.

5 Simulation and Plotting from MDT

Simulation and plotting is possible from a special command window, where commands are sent to `omc`. For example:

Simulation:

```
>> simulate(Influenza,startTime=0.0,
stopTime=3.0)
record
    resultFile = "Influenza_res.plt"
end record
```

The simulated population is plotted (Figure 11).

```
>> plot({Infected_Popul.p})
true
```

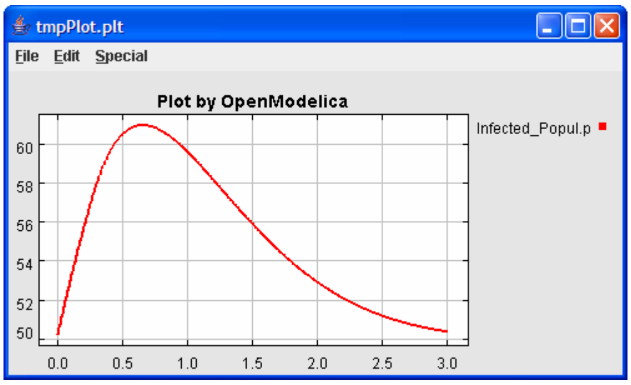


Figure 11. Plot of the Influenza model.

6 Eclipse History

In the mid 1990s software developments tools were primarily dominated by systems built around two technologies. Many of the tools were focused on a runtime environment developed and controlled by the Microsoft Corporation. The other was built around the Java platform. The Java platform is less dominated 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 become independent of Microsoft.

By creating a common platform for development tools built on top of the Java platform, IBM hoped to attract more developers from competing environments. 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, the work on a new Java IDE was performed at the IBM laboratories. At the same time additional teams were setup by IBM to build other products on top of the platform.

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 the 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 still perceived 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 has 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 adoption and recognition than any earlier versions. Today (2005) 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 foundation 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 strong foothold in the industry and is one of the major open source software development platforms [2].

7 Conclusion

The OpenModelica integrated development environment for Modelica has been augmented with a plugin to the Eclipse framework. The plugin, called MDT (Modelica Development Tooling) [13], an earlier version is described in [14], is primarily aimed at development of large models. It has support for browsing, editing, code completion, automatic indentation, building executables, and debugging. It also allows simulation and plotting from a special command window. To summarize, it provides a rather complete integrated development environment, and it is also the first available Eclipse plugin for an equation-based language.

8 Acknowledgements

This work was supported by Vinnova in the SWEB-Prod project, by the CUGS graduate school, and by MathCore Engineering AB.

References

[1] Eclipse website. http://www.eclipse.org.

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

[3] Peter Fritzson. Symbolic Debugging through Incremental Compilation in an Integrated Environment, *Journal of Systems and Software*, 3, pp. 285–294, 1983.

[4] Peter Fritzson, Peter Aronsson, Håkan Lundvall, Kaj Nyström, Adrian Pop, Levon Saldamli, David Broman. The OpenModelica Modeling, Simulation, and Development Environment. In *Proceedings of the 46th Conference on Simulation and Modelling of the Scandinavian Simulation Society* (SIMS2005), Trondheim, Norway, October 13-14, 2005. http://www.ida.liu.se/projects/OpenModelica

[5] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*, 940 pp., ISBN 0-471-471631, Wiley-IEEE Press, 2004.

[6] The Modelica Association. The Modelica Language Specification Version 2.2, March 2005. http://www.modelica.org.

[7] Peter Fritzson et al. The OpenModelica Users Guide, version 0.7, May 2006. http://www.ida.liu.se/projects/OpenModelica.

[8] Peter Fritzson et al. The OpenModelica System Documentation, version 0.7, May 2006. http://www.ida.liu.se/projects/OpenModelica.

[9] J. Lindskov, M. Knudsen, O. Löfgren, Ole Lehrmann-Madsen, and Boris Magnusson (Eds.). Object-Oriented Environments - The Mjølner Approach. Prentice Hall, 1993.

[10] Adrian Pop and Peter Fritzson. A Portable Debugger for Algorithmic Modelica Code, the 4th International Modelica Conference (Modelica2005), March 7-9, 2005, Hamburg, Germany.

[11] Erik Sandewall. Programming in an Interactive Environment: The "LISP" Experience, *Computing Surveys*, 10:1, Mar. 1978.

[12] Warren Teitelman. *INTERLISP Reference Manual*. Xerox Palo Alto Research Center, Palo Alto, CA, 1974.

[13] PELAB, *Modelica Development Tooling (MDT).* http://www.ida.liu.se/labs/pelab/modelica/OpenModelica/MDT

[14] Andreas Remar and Elmir Jagudin. Modelica Development Tooling for Eclipse. Master Thesis LITH-IDA-EX–06/024–SE, April 10, 2006.