Institutionen för datavetenskap
Department of Computer and Information Science

Final thesis

Design and Implementation of a User Friendly OpenModelica Graphical Connection Editor

by

Syed Adeel Asghar & Sonia Tariq

LIU-IDA/LITH-EX-A--10/047--SE

2010-12-08

Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings universitet
581 83 Linköping

Examiner
Prof. Peter Fritzson

Supervisor
Dr. Mohsen Torabzadeh-Tari

Advisor
Mr. Martin Sjölund

Table of Contents

Dedication

We dedicate our work to our beloved parents, respectable teachers, and to the university. Without their prayers, love, support and encouragement we would not be able to accomplish this task.

Acknowledgements

First of all we are thankful to Almighty Allah who gave us life, knowledge and strength to attest our skills. Our deepest thanks go to our supervisor Dr. Mohsen Torabzadeh-Tari, and our examiner Prof. Peter Fritzson who guided us through the bulk of the work. We express gratitude to our associated supervisor, Mr. Martin Sjölund, as his detailed and constructive comments provided vital guidance for our thesis development.

We would also like to say thanks to our friends, colleagues, and specially Dr. Adrian Pop, as their contributions are immense in nipping errors and cleaning up the discussions.

We would like to extend our appreciation to IEI, Department of Management and Engineering, Linköping University who provided us a GUI prototype called HOPSAN which eventually become the starting point of our thesis. And lastly, we thank Mr. Filippo Donida for explaining some issues related to Modelica annotations and SimForge.

Abstract

OpenModelica (www.openmodelica.org) is an open-source Modelica-based modeling and simulation environment intended for industrial as well as academic usage. Its long-term development is supported by a non-profit organization – the Open Source Modelica Consortium OSMC, where Linköping University is a member.

The main reason behind this thesis was the need for a user friendly, efficient and modular OpenModelica graphical connection editor. The already existing open source editors were either textual or not so user friendly. As a part of this thesis work a new open source Qt-based cross platform graphical user interface was designed and implemented, called OMEdit, partially based on an existing GUI for hydraulic systems, HOPSAN. The usage of Qt C++ libraries makes this tool more future safe and also allows it to be easily integrated into other parts of the OpenModelica platform.

This thesis aims at developing an advanced open source user friendly graphical user interface that provides the users with easy-to-use model creation, connection editing, simulation of models, and plotting of results. The interface is extensible enough to support user-defined extensions/models. Models can be both textual and graphical. From the annotation information in the Modelica models (e.g. Modelica Standard Library components) a connection tree and diagrams can be created. The communication to the OpenModelica Compiler (OMC) Subsystem is performed through a Corba client-server interface. The OMC Corba server provides an interactive API interface. The connection editor will function as the front-end and OMC as the backend. OMEdit communicates with OMC through the interactive API interface, requests the model information and creates models/connection diagrams based on the Modelica annotations standard version 3.2.

Keywords: Graphic editor, connection diagrams, Modelica, modeling, simulation, OpenModelica.

List of Acronyms and Abbreviations

API	Application Programming Interface
AWT	Abstract Windowing Toolkit
Corba	Common Object Request Broker Architecture
IOR	Interoperable Object Reference
MSL	Modelica Standard Library
OMC	OpenModelica Compiler
OMNotebook	OpenModelica Electronic Notebook
OMShell	OpenModelica Shell
OMI	OpenModelica Interactive
IDL	Interface Definition Language
IDE	Integrated Development Environment
DS	Dassault Systèmes
URL	Uniform Resource Locator
JVM	Java Virtual Machine
DLR	Deutsches Zentrum für Luft- und Raumfahrt

List of Figures

List of Tables

Overview

Chapter 1:

This chapter describes the background information about the object-oriented Modelica language and its different versions, and the open source OpenModelica platform. Also the purpose behind the thesis is explained here together with an analysis study done in the design stages of this project.

Chapter 2:

This chapter elaborates all the requirements of the thesis.

Chapter 3:

Chapter 3 describes all the development tools used while creating the application.

Chapter 4:

Chapter 4 starts with the introduction to OMEdit - OpenModelica Connection Editor and then moves forward with the demonstration of an introductory model in OMEdit.

Chapter 5:

This chapter gives a detailed walkthrough of OMEdit. Including information about all the dialogs and windows, and also briefly explains what functionality is each interface is performing.

Chapter 6:

This chapter contains one of the major parts of the report. Chapter 6 discusses the design and implementation of OMEdit. It highlights the OMC communication mechanism details and the Modelica annotations description.

Chapter 7:

Chapter 7 lists the related model editors that are available open source as well as commercial.

Chapter 8:

This chapter briefly discusses some of the future implementations of OMEdit. The future milestones like integrated OMNotebook, interactive simulation and integrated OMShell.

Appendix A:

It contains the source code of the `DCmotor` model that is being used in Chapter 4 section 4.3 for demonstrating that how a model is created in OMEdit.

Appendix B:

It lists some of the OMC API commands, however there is huge number of OMC API commands and all commands are not listed here.

Chapter 1 Introduction

- An overview of Modelica.
- History of Modelica.
- Features of Modelica language.
- A brief introduction of OpenModelica.
- What is the purpose behind this master thesis?
- How has the study and analysis been carried out?

1.1 Modelica - Introduction

Modelica is a freely available, object-oriented language for modeling of large, complex, and heterogeneous physical systems. It is suited for multi-domain modeling, for example, mechatronic models in robotics, automotive and aerospace applications involving mechanical, electrical, hydraulic and control subsystems, process oriented applications and generation, and distribution of electric power. Modelica is designed such that it can be utilized in a similar way as an engineer builds a real system: First trying to find standard components like motors, pumps and valves from manufacturers catalogues with appropriate specifications and interfaces and only if there does not exist a particular subsystem, a component model would be newly constructed based on standardized interfaces [1].

Models in Modelica are mathematically described by differential, algebraic and discrete equations. No particular variable needs to be solved for manually. A Modelica tool will have enough information to decide that automatically. Modelica is designed such that available, specialized algorithms can be utilized to enable efficient handling of large models having more than a hundred thousand equations [1].

Modelica is a free language. Its design has been influenced by many object oriented modeling languages. The Modelica community works hard and gives their precious time to provide many services for their users like newsletters, free educational materials, lots of freely downloadable papers and software, training courses, job offerings, and student work. There are a lot of people involved in the design of this language, however a couple of key architects can be mentioned, Peter Fritzson from Linköping university, Martin Otter from DLR (Deutsches Zentrum für Luft- und Raumfahrt), and Hilding Elmqvist from Dynasim (now DS (Dassault Systèmes)). The basic aim was to develop an efficient object-oriented modeling language for modeling technical systems for reprocess and exchange of models of dynamic systems in a standardized format. Modelica helps several automotive companies in designing their energy efficient vehicles furthermore, it also facilitate to enhanced air conditioning systems etc.

1.2 Modelica - Versions

The following table shows the Modelica versions release over the years,

Version No.	**Description**	**Release Date**
1.0	This version is based on DAE system with some discrete features to handle discontinuities and sampled systems.	September, 1997
1.1	Version 1.1 introduced the prefixes discrete and non-discrete (pre, when) features, array expression for semantics, lots of built-in functions and operators.	December, 1998
1.2	This version allows code optimization, how to interface with C and FORTRAN, semantics for if-clauses, fixed and nominal attributes dynamic types of inner/outer. Also provide the higher flexibility about changing the external function interface.	15th June, 1999
1.3	This version introduced the connection semantics for inner/outer connectors, semantics for protected element, and the scope of for-loop variables and improved array expressions.	December, 1999
1.4	The main changes in this version are:	December, 2000

	Refined packages, and refined if and when-clauses, functions can specify their derivative and many more.	
2.0	Support for generic formulation of blocks applicable to scalar and vector connectors, introduced enumeration types, specified the graphical appearance of Modelica object diagrams etc.	July, 2002
2.1	New annotations for version handling of libraries and models, for library specific error messages, etc. Introduced `break` and `return` statements in while loop and function respectively. Also many other enhancements.	March, 2004
2.2	The concept of Expandable connector was launched, recursive inner/outer definitions etc.	February, 2005
3.0	All the previous versions of Modelica are more or less backward compatible with minor exceptions but this is the first version which is slightly non-backwards compatible compared to earlier versions. It has almost all the previous features, but also imposes the new constraint of balanced model checking for earlier error detection. This version is also called a `clean-up` version.	September, 2007
3.1	This version brings up several new features like URI's support in Modelica documentation annotation, better support for expandable connectors, introduction of stream connectors, overloaded operators, etc.	May, 2009
3.2	Version 3.2 provides better support for object libraries, access control to protect intellectual property, functions as formal inputs to functions etc.	March, 2010

Table 1-1: Modelica versions [14]

1.3 Features of Modelica

Modelica supports high-level models for composition and detailed modeling of library component. Models of standard components are usually collected into libraries of models. With a graphical model editor a physical system can be modeled by simply drawing a connection diagram, including positioning of the models representing the components, making connections, and entering parameter values in the dialog boxes.

1.4 OpenModelica

OpenModelica is an open source equation based modeling and simulation environment intended both for industrial as well as academic usage [2]. The primary target language is Modelica; however it is not only restricted to Modelica, C and FORTRAN code can also be compiled. More recently UML software modeling integrated with Modelica is being supported through the ModelicaML UML/Modelica profile The OpenModelica project is coordinated by PELAB, the Programming Environments Laboratory, at Linköping University [3].

1.5 Purpose

OpenModelica has previously only had SimForge from Politecnico di Milano [4], for creating model diagrams. Although it provides most of the required functionalities needed in any connection editing tool, it lacks responsiveness to the user (i.e., slow), as well as being a bit unstable, difficult to use for the beginner, and not providing good support for Modelica 3.1 graphical annotations in the MSL 3.1 library. Therefore, a new advanced user interface was required to provide all the features that are left out in SimForge. See Chapter 7 section 7.1 for a detailed description of why a new graphical user interface was needed.

The main purpose of developing the new OpenModelica graphical editor was to provide a free user friendly environment for its users. The new interface overcomes the present problems and difficulties which users were facing while creating the models, connecting the instances, speed etc.

1.6 Study and Analysis Phase

When you start some new work you should have good knowledge about the problem area. Therefore, in order to understand the thesis requirements, we studied many papers about Modelica and OpenModelica and lots of other documents which helped us to understand the requirements. Before starting the implementation we learned about the previous work (SimForge), how it was functioning, how the application was built, what the problems and their consequences were, how we could avoid the design mistakes that previous tools had done, and what could help us etc. The most time consuming task in this thesis work was to learn how SimForge as well as commercial tools like Dymola [5] and MathModelica [6] worked. We had to study the Modelica specifications in order to understand the requirements and then we mapped it to the tools and verify how they had implemented it.

A five days workshop organized by PELAB under the supervision of Prof. Peter Fritzson about OpenModelica also helped us in gaining knowledge and solving problems.

The design and implementation of this editor used as a starting point an already existing GUI prototype structure for hydraulic systems, the Hopsan GUI developed by IEI [7] Department of Management and Engineering at Linköping University.

Chapter 2 Requirements

- An advanced and easy-to-use responsive graphical user interface (GUI) was needed.
- The GUI should allow Modelica Standard Library browsing, preferably with the latest version MSL 3.1.
- Modelica component models including graphical annotations should be interpreted and displayed as graphical objects.
- Users are allowed to create shapes of their own choice.
- The application should use the OpenModelica simulation and plotting mechanism.

2.1 Advanced User Interface

As mentioned earlier, there already exists an open source graphical model editor called SimForge that can be used with OpenModelica. But due to performance and stability issues discussed in Chapter 7 section 7.1, a new advanced user interface was needed. The Qt platform, i.e., the C++ open source Qt libraries, was chosen for the GUI development for several reasons. Both of the commercial tools Dymola and MathModelica are using Qt for their GUI development, showing that it is possible to create a highly performing and stable GUI for a graphical connection editor based on Qt. Moreover, the Hopsan GUI showed that it was possible to create a fast and functioning connection editor (although simpler than a Modelica counterpart) with a rather limited effort. Third, the Qt platform supports portable GUIs between the main platforms Windows, Linux, and Mac. Fourth, the OpenModelica OMNotebook GUI is already implemented using Qt, which simplifies integration between the connection editor and notebook GUI. Also the fact that the group developing Hopsan offered to share their Qt GUI did lower the design and implementation effort for a new GUI.

2.2 Modelica Standard Library Browsing

The GUI should support automatic loading of the Modelica Standard Library and make it available to the user for browsing. The user should be able to drag and drop the pre-defined component models into the connection diagram.

2.3 Pre-defined Component Models as Shapes

The GUI should create and display the shape of each pre-defined component model based on the Modelica graphical annotations standard 3.2. The pre-defined components contain shape information, i.e. annotations that OMEdit uses to draw the component shapes in the graphical view. See Chapter 6 section 6.2.1.

2.4 User Defined Shapes

The application should allow users to create shapes, also called icons, for their own created models. The shapes should follow the Modelica annotations 3.2 standard. Read more details about this in Chapter 4 section 4.4

2.5 Simulation

The interface should be able to simulate a model created by the user using the simulation mechanism provided by OpenModelica.

2.6 Plotting

The simulation of a model creates a simulation result file which contains a list of model instance variables that are candidates for plotting. The GUI should use the existing OpenModelica plotting feature to plot the variables, or if time permitted replace the current OpenModelica plotting mechanism with a better plotting package.

Chapter 3 Tools and Communication

- The OMC interactive API is used for communication with OMC.
- OmniORB is used to implement the OMC communication module.
- OMEdit is developed using C++ and the Qt libraries.
- The GUI is created using the Qt libraries.
- Qt Creator is used as the main development environment.
- The source code documentation is done using Doxygen.

3.1 OpenModelica Compiler Interactive API

The OpenModelica Compiler (OMC) Interactive API is used in order to communicate with OMC. The communication is carried out through the Corba interface provided by the OpenModelica Compiler. See Chapter 6 section 6.1 for the detailed description of OMC communication.

3.2 OmniORB

OMEdit acts as a client in the OMC communication mechanism. The Corba client in OMEdit is based on the OmniORB Corba implementation [13]. The OmniORB IDL (Interface Definition Language) compiler takes the OMC IDL file as input and creates the stub files.

IDL defines the interface for a software in a language neutral way. The stub files generated by the IDL handle the communication between the two programs, which can be written in two different languages.

The stub files created using the OMC IDL file are used by OMEdit to send/receive commands. See Chapter 6 section 6.1 for further details.

3.3 C++

OMEdit is primarily developed in C++. It uses the Mingw [8] compiler for Windows executables and GCC [9] for the Linux versions.

3.4 Qt Framework

Qt is a cross platform application and user interface framework [11]. The cross platform feature of Qt is one of the main features that made us adopt Qt framework, the huge and enriched Qt graphics library well-proven by several professional tools is the other.

OMEdit uses Qt for the GUI development. The main Qt features used are briefly describes below,

- *Qt Graphics View Framework* – The `Designer Window` (see Chapter 5 section 5.1.2) is based on the `QGraphicsView` and `QGraphicsScene` classes. The Graphics view provides the surface for creating and managing 2 dimensional graphical items. It also provides features like scaling, rotating, and transforming graphical items. The `QGraphicsItem` class is used to create component model shapes based on the Modelica annotations 3.2 standard.
- *Qt Webkit Module* – The `Documentation Window` (see Chapter 5 section 5.1.5) is based on Qt's Webkit module. The Webkit module provides classes that are used to render `HTML` contents. The Webkit module contains a `QWebView` class which is used to display the Modelica documentation annotations.
- *Qt Widgets & Dialogs* – The Qt's widgets (*in this report widgets are referred to by the term windows*) and dialogs are used throughout the application to display contents, information messages, forms etc. (see Chapter 5 for more details).

3.5 Qt Creator

Qt Creator is a cross platform Integrated Development Environment (IDE), used for the development of OMEdit. The other possible candidate was Visual Studio 2010 [12], but in order to use Visual Studio 2010 a separate add-in installation was required. Although, installing the add-in is not a big task and was not the primary reason for not using Visual Studio 2010. The main reason behind not

using Visual Studio 2010, one of the most robust IDEs available in market, was that it lacks the Qt's `IntelliSense` feature; the feature that actually is available in Visual Studio 2010 doesn't sometimes show the declarations which are actually present. Compared to Visual Studio 2010, Qt Creator is very strong with Qt's `IntelliSense` because it is primarily created with this feature, and gives the rapid application development environment to the user.

3.6 Doxygen

Doxygen is a documentation system used for a lot of languages including C++ [10]. For the source code documentation of OMEdit we used Doxygen.

Chapter 4 Getting Started

- A brief introduction of OMEdit
- How to start OMEdit?
- How to create a `DCmotor` model in OMEdit?
- How to create user defined shapes in OMEdit?

4.1 About OMEdit

OMEdit – the OpenModelica Connection Editor is the new Graphical User Interface for graphical model editing in OpenModelica. It is implemented in C++ using the Qt 4.7 [11] graphical user interface library and supports the Modelica Standard Library version 3.1 that is included in the latest OpenModelica (version 1.6.0) installation. This chapter gives a brief introduction to OMEdit and also demonstrates how to create a `DCmotor` model using it.

OMEdit provides user friendly features like;

- *Modeling* – Easy Modelica model creation.
- *Pre-defined models* – Browsing the Modelica Standard Library to access the provided models.
- *User defined models* – Users can create their own models for immediate usage and later refinement and reuse.
- *Component interfaces* – Smart connection editing for drawing and editing connections between model interfaces.
- *Simulation subsystem* – Subsystem for running simulations and specifying simulation parameters start and stop time, etc.
- *Plotting* – Interface to plot variables from simulated models.

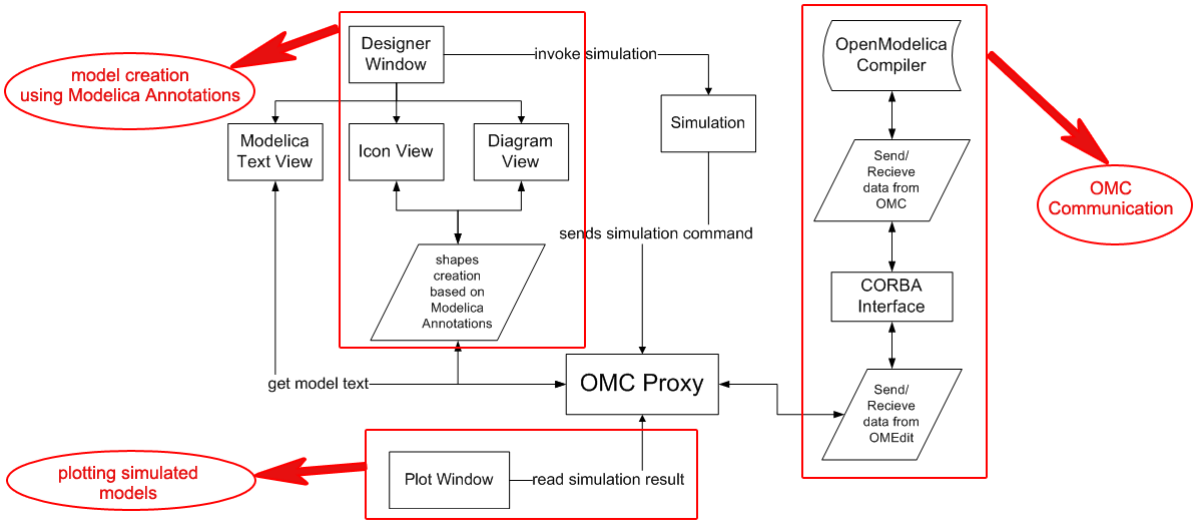


Figure 4-1: OMEdit high level view

OMEdit uses the OmniORB CORBA implementation [13] to communicate with the OpenModelica Compiler. The Modelica 3.2 Graphical Annotations [14] present in most models are interpreted for drawing Modelica Standard Library component models and user defined models. As a result, the interoperability with other Modelica tool vendors becomes easier as the Modelica icon and diagrams defined in other tools supporting the Modelica 3.1 or Modelica 3.2 standards are easily handled in OMEdit. OMEdit also uses annotations for displaying Modelica documentation. See Figure 5-2.

4.2 How to Start OMEdit?

OMEdit can be launched using the executable placed in `$OPENMODELICAHOME/bin`. A splash screen similar to the one shown in Figure 4-2 will appear indicating that it is starting OMEdit. After the splash screen the main OMEdit window will appear; see Figure 4-3.

Figure 4-2: OMEdit splash screen

4.3 Introductory Model in OMEdit

In this section we will demonstrate how one can create Modelica models in OMEdit, e.g. a `DCmotor`.

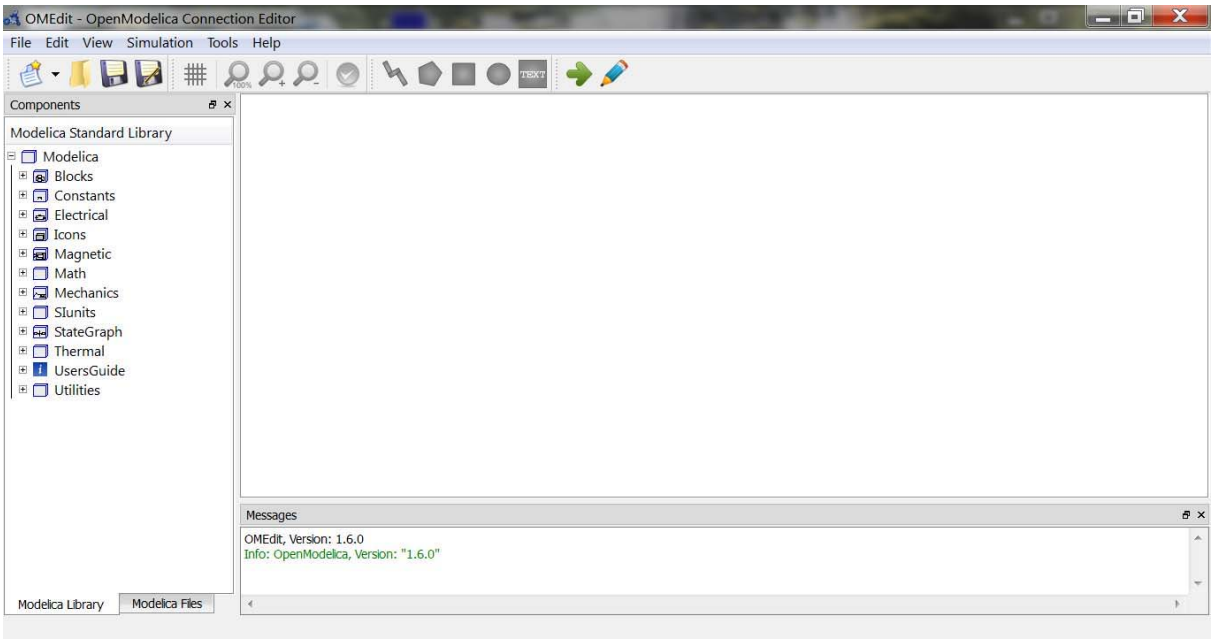


Figure 4-3: OMEdit Main Window

4.3.1 Creating a New File

Creating a new file/model in OMEdit is rather straightforward. In OMEdit the new model can be of type `model`, `class`, `connector`, `record`, `block`, `function` or `package`. The user can create any of the model types mentioned above by selecting `File > New` from the menu. Alternatively, you can also click on the drop down button beside `new icon` shown in the toolbar right below the File menu. See Figure 4-5.

In this introductory example we will create a new model named `DCmotor`. By default the newly created model will open up in the tabbed view of OMEdit, also called `Designer Window` (see Chapter 5 section 5.1.2), and become visible. The models are created in the OMC global scope unless you specify the parent package for it.

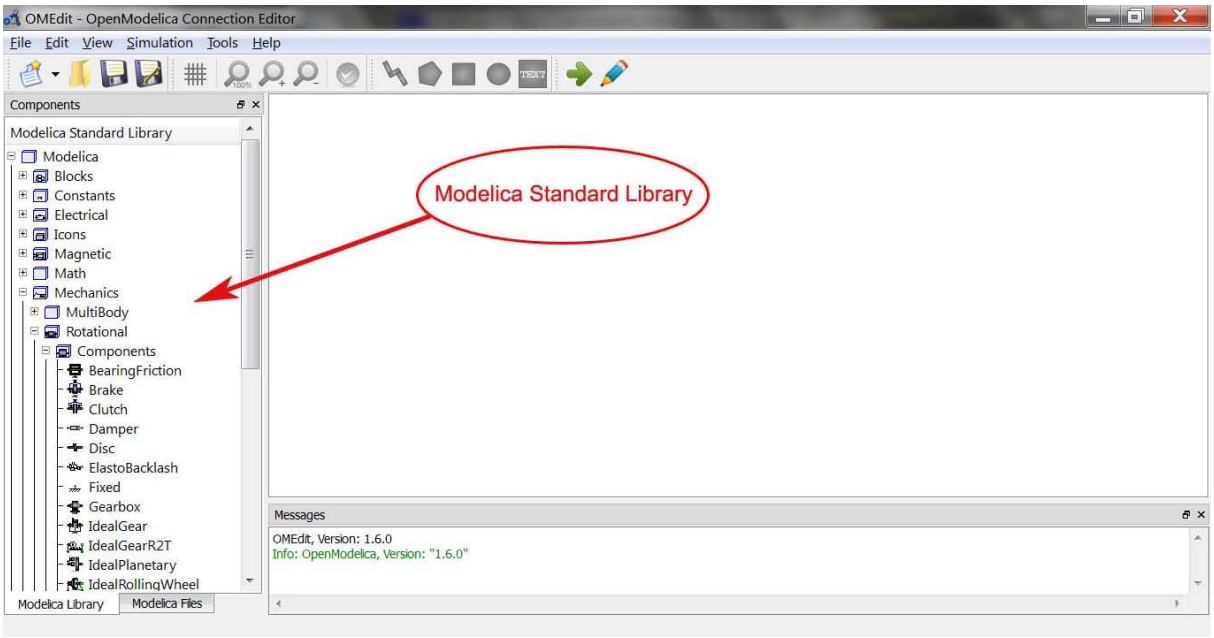


Figure 4-4: Modelica Standard Library

4.3.2 Adding Component Models

The Modelica standard library is loaded automatically and is available in the left dock window. The library is retrieved through the `loadModel(Modelica)` API call and is loaded into the OMC symbol table and workspace after the command execution is completed. Instances of component models available in the Modelica Standard Library can be added to the currently edited model by doing a drag and drop from the `Library Window` (see Chapter 5 section 5.1.1). Navigate to the component model in the library tree, click on it, drag it to the model you are building while pressing the mouse left button, and drop the component where you want to place it in the model.

For this example we will add four components as instances of the models `Ground`, `Resistor`, `Inductor` and `EMF` from the `Modelica.Electrical.Analog.Basic` package, an instance of the model `SignalVoltage` from the `Modelica.Electrical.Analog.Sources` package, one instance of the model `Inertia` from the `Modelica.Mechanics.Rotational.Components` package and one last instance of the model `Step` from the `Modelica.Blocks.Sources` package.

4.3.3 Making Connections

In order to connect one component model to another the user simply clicks on any of the ports. Then it will start displaying a connection line. Then move the mouse to the target component where you want to finish the connection and click on the component port where the connection should end. You do not need to hold the mouse left button down for drawing connections.

In order to have a functioning `DCmotor` model, connect the `Resistor` to the `Inductor` and the `SignalVoltage`, `EMF` to `Inductor` and `Inertia`, `Ground` to `SignalVoltage` and `EMF`, and finally `Step` to `SignalVoltage`. Check Figure 4-6 to see how the `DCmotor` model looks like after connections.

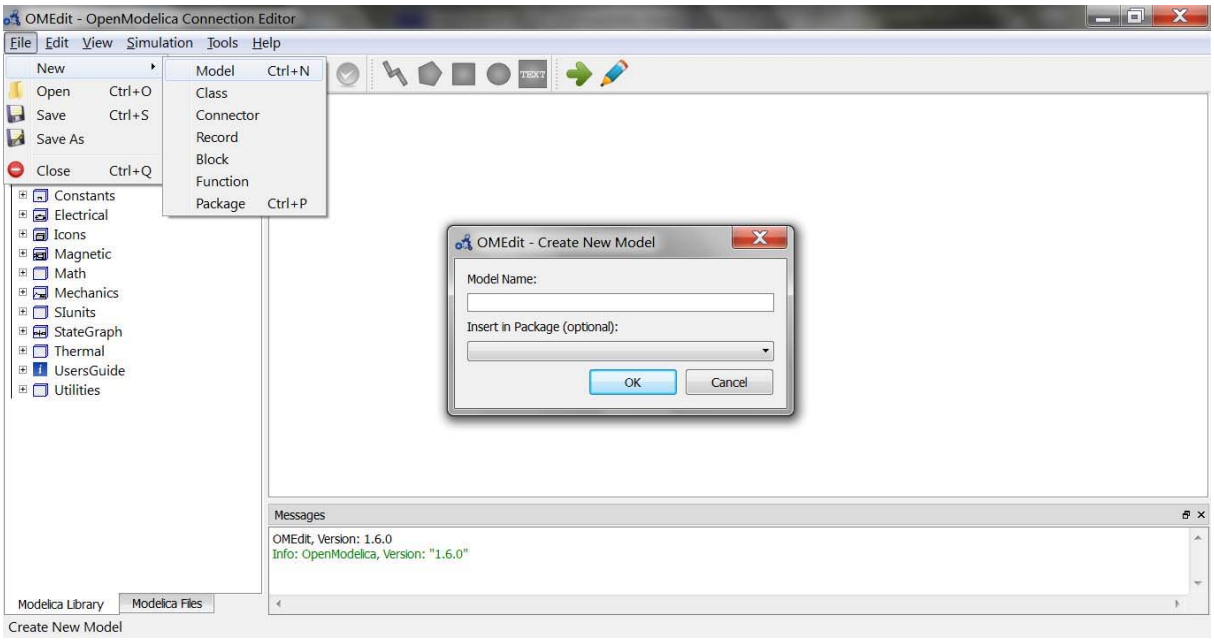


Figure 4-5: Creating a new model

4.3.4 Simulating the Model

The OMEdit Simulation Center dialog (see Chapter 5 section 5.2.2) can be launched either from `Simulation > Simulate` or by clicking the `simulate icon` from the toolbar. Once the user clicks on `Simulate!` button, OMEdit starts the simulation process. At the end of the simulation process the Plot Variables Window (see Chapter 5 section 5.1.3) useful for plotting will appear at the right side. Figure 4-7 shows the simulation dialog.

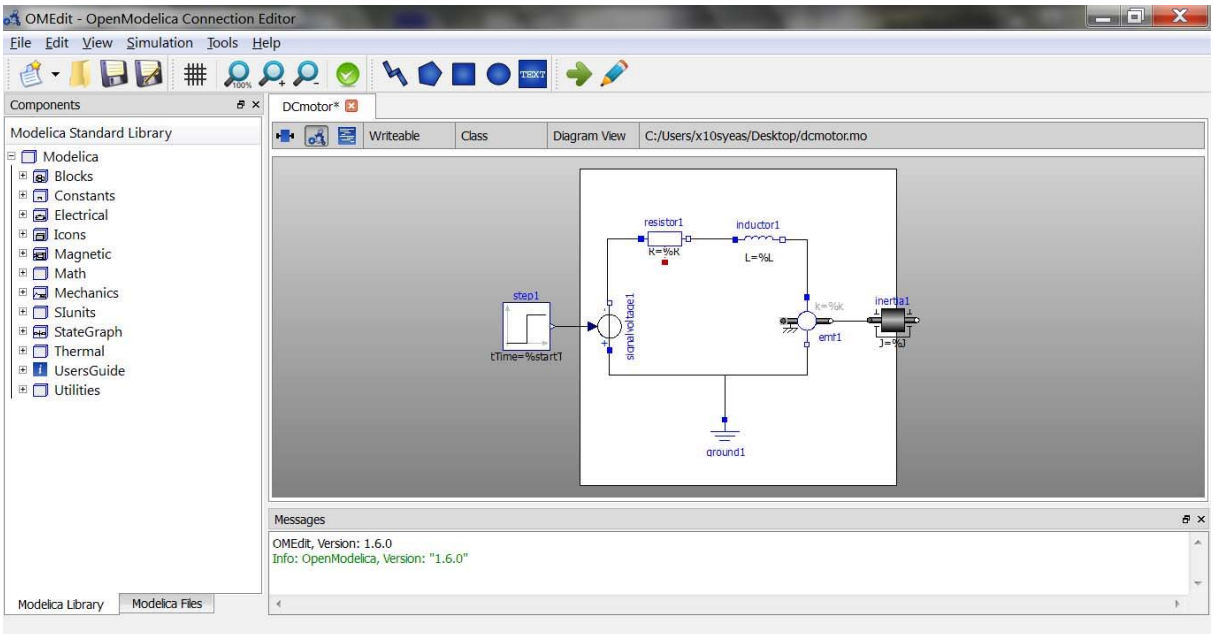


Figure 4-6: DCmotor model after connections

4.3.5 Plotting Variables from Simulated Models

The instance variables that are candidates for plotting are shown in the right dock window, see Figure 4-8. This window is automatically launched once the user simulates the model; the user can

also launch this window manually either from `Simulation > Plot Variables` or by clicking on the `plot` icon from toolbar. It contains the list of variables that are possible to use in an OpenModelica plot. The plot variables window contains a tree structure of variables; there is a checkbox beside each variable. The user can launch the plotted graph window by clicking the checkbox.

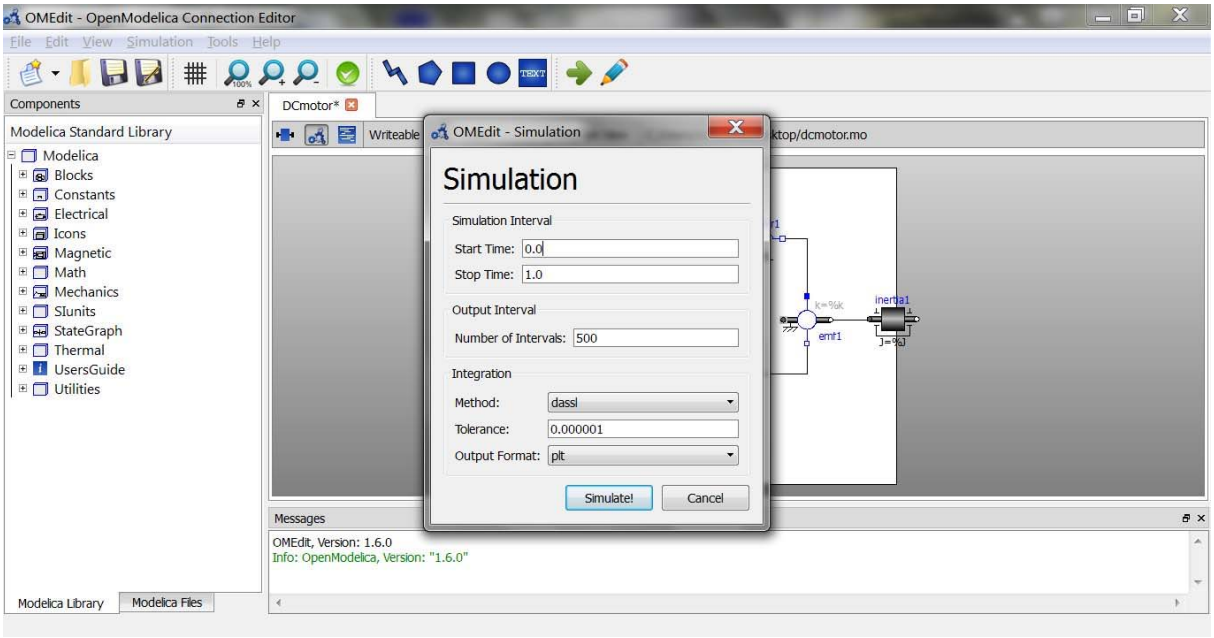


Figure 4-7: Simulation Dialog

Figure 4-8 shows the complete `DCmotor` model along with the list of plot variables and an example plot window.

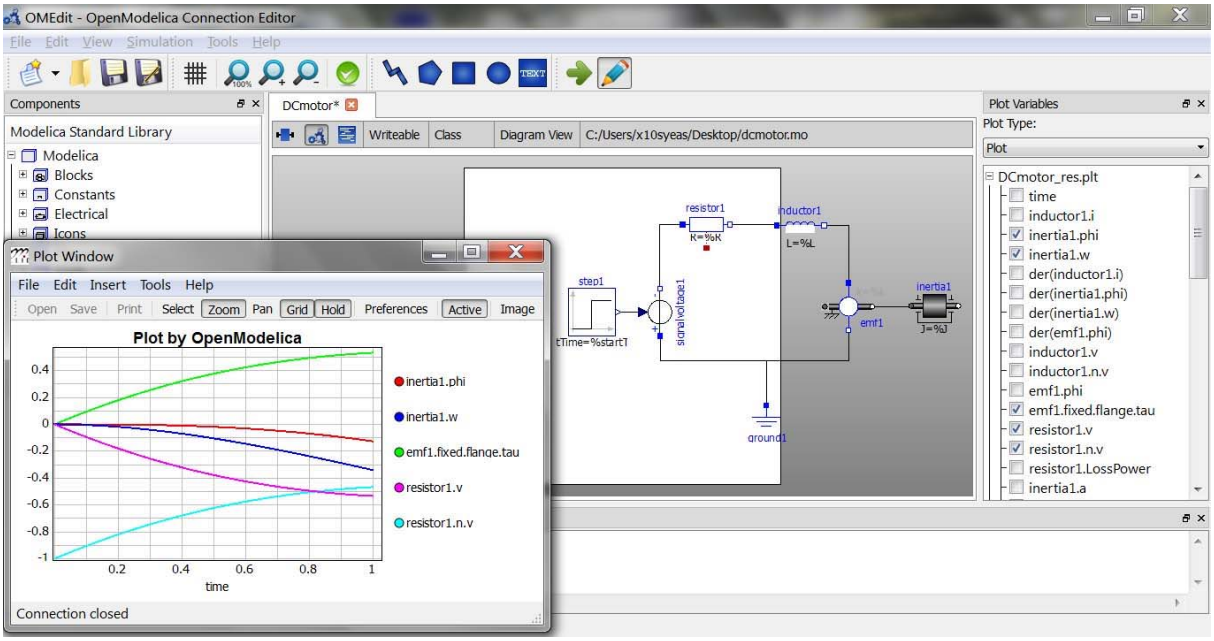


Figure 4-8: Plotted Variables

4.4 How to Create User Defined Shapes/Icons?

The user can also create shapes by using the 6 types of shape tools available in OMEdit.

- *Line Tool* – draws a line. A line is created with a minimum of two points. In order to create a line, the user first selects the line tool from the toolbar and then click on the `Designer`

Window; this will start creating a line. If a user clicks again on the Designer Window a new line point is created. In order to finish the line creation, user has to double click on the Designer Window.

- *Polygon Tool* – draws a polygon. A polygon is created in a similar fashion as a line is created. The only difference between a line and a polygon is that if a polygon contains two points it will look like a line and if a polygon contains more than two points it will become a closed polygon shape.
- *Rectangle Tool* – draws a rectangle. The rectangle only contains two points where the first point indicates the starting point and the second point indicates the ending point. In order to create a rectangle, the user has to select the rectangle tool from the toolbar and then click on the Designer Window, this click will become the first point of rectangle. In order to finish the rectangle creation, the user has to click again on the Designer Window where he/she wants to finish the rectangle. The second click will become the second point of rectangle.
- *Ellipse Tool* – draws an ellipse. The ellipse is created in a similar way as a rectangle is created.
- *Text Tool* – draws a text label.
- *Bitmap Tool* – draws a bitmap container.

The shape tools are located at the top in the toolbar. See Figure 4-9.

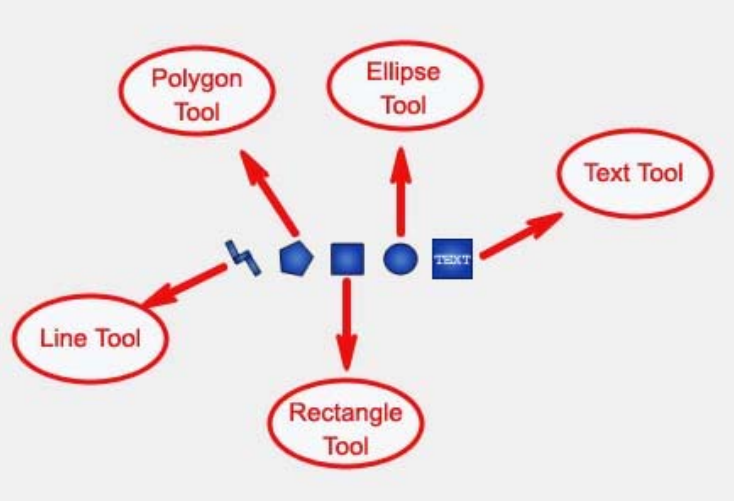


Figure 4-9: User defined shapes

The user can select any of the shape tools and start drawing on the Designer Window. The shapes created on the Diagram View of Designer Window are part of the diagram and the shapes created on the Icon View will become the icon representative of the model.

For example, if a user creates a model with name testModel and add a rectangle using the rectangle tool and a polygon using the polygon tool, in the Icon View of the model. The model's Modelica Text will look like,

```
model testModel
annotation(Icon(graphics = {Rectangle(rotation = 0, lineColor = {0,0,255},
fillColor = {0,0,255}, pattern = LinePattern.Solid, fillPattern =
FillPattern.None, lineThickness = 0.25, extent = {{ -64.5,88},{63, -
22.5}}),Polygon(points = {{ -47.5, -29.5},{52.5, -29.5},{4.5, -86},{ -47.5, -
29.5}}, rotation = 0, lineColor = {0,0,255}, fillColor = {0,0,255}, pattern =
LinePattern.Solid, fillPattern = FillPattern.None, lineThickness = 0.25)}));
end testModel;
```

In the above code snippet of testModel, the rectangle and a polygon are added to the icon annotation of the model. Similarly, any user defined shape drawn on a Diagram View of the model will be added to the diagram annotation of the model.

At the time of finalizing this thesis the icon editor described here is not completely implemented. However a rather small amount of work remains, and it is expected to be finalized soon.

Chapter 5 OMEdit Windows and Dialog Boxes

- Library Window for Modelica Standard Library.
- Drawing interface in the form of Designer Window.
- Plot Window contains the list of instance variables.
- Messages Window displays the information, warning and error messages.
- Documentation Window displays the Modelica annotations based documentation in a QWebView.
- New Dialog for creating Modelica models.
- Simulation Dialog for simulating Modelica models.

5.1 Windows

OMEdit displays a number of windows that show different views to users.

5.1.1 Library Window

The Modelica Standard Library (MSL) is automatically loaded into OMEdit. An entry for the MSL is located on the left dock window. Once a new Modelica model has been started then the user can just drag and drop components from the MSL the `Library Window` into the model. The available libraries in the MSL are:

- Blocks
- Constant
- Electric
- Icons
- Magnetic
- Math
- Mechanics
- SIunits
- Thermal
- UsersGuide
- Utilities

The `Library Window` consists of two tabs. One shows the Modelica Standard Library and is selected by default. The other tab shows the Modelica files that user creates in OMEdit.

5.1.1.1 Viewing a Model's Description

In order to view the model details, double click the component and details will be opened in the `Designer Window`. An alternative way is to right click on the component and press `Show Component`, this will do the same.

5.1.1.2 Viewing a Model's Documentation

Right click the model in the `Library Window` and select `View Documentation`; this will launch the `Documentation Window`. See Figure 5-1.

5.1.1.3 How to Check a Model?

Right click the component in the library window and select ***Check***; this will launch the ***Check Dialog***. See Figure 5-1.

5.1.1.4 How to Rename a Model?

Right click the model in the `Library Window` and select `Rename`; this will launch the `Rename Dialog`. See Figure 5-1.

5.1.1.5 How to Delete a Model?

Right click the model in the `library window` and select `Delete`; a popup will appear asking "Are you sure you want to delete?"

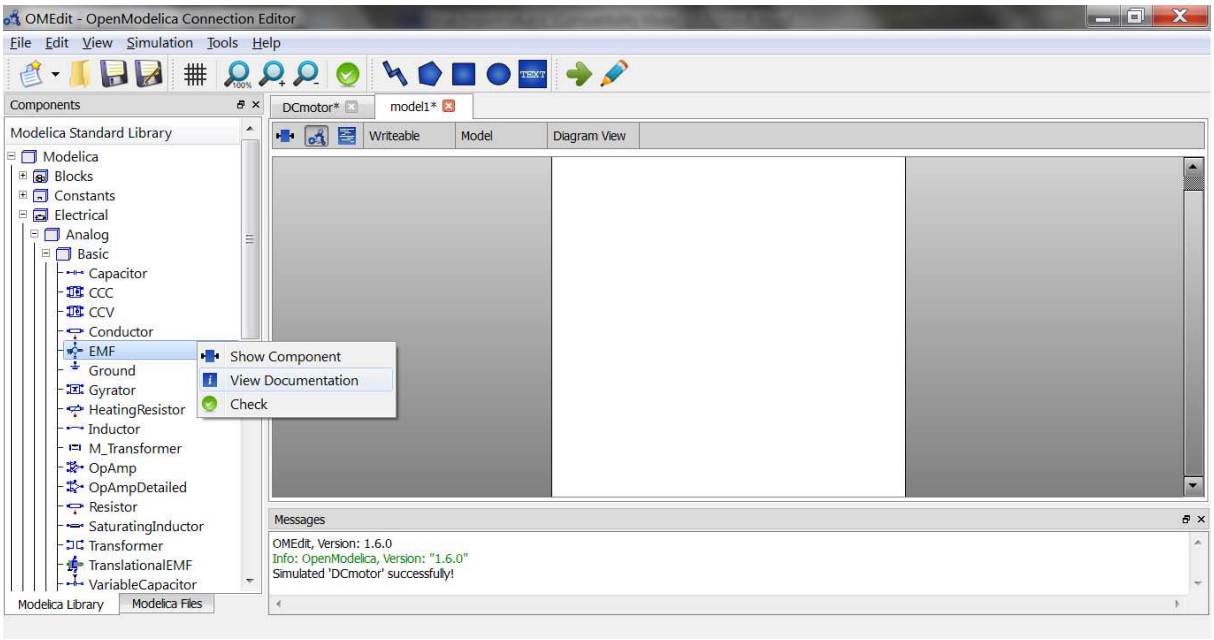


Figure 5-1: Context menu to view component model details

5.1.2 Designer Window

The designer Window is the main window of OMEdit. It consists of three views,

- *Icon View* - Shows the model icon view.
- *Diagram View* - Shows the diagram of the model created by the user.
- *Modelica Text View* - Shows the Modelica text of the model.

5.1.3 Plot Window

The right dock window represents the `Plot Window`. It consists of a tree containing the list of instance variables that are extracted from the simulation result. Each item of the tree has a checkbox beside it. The user can click on the check box to launch the plot graph window. The user can add/remove the variables from the plot graph window by marking/unmarking the checkbox beside the plot variable.

5.1.4 Messages Window

The `Messages Window` is located at the bottom of the application and consists of 4 types of messages,

- *General Messages* – Shown in black color.
- *Informational Messages* – Shown in green color.
- *Warning Messages* – Shown in orange color.
- *Error Messages* – Shown in red color.

5.1.5 Documentation Window

This window is shown when a user right clicks the model component in the library window and selects `View Documentation`. This shows the OpenModelica documentation of models in a web view. All externals links present in the documentation window are opened in the default browser of

the user. All local links are opened in the same window. Figure 5-2 shows the `Documentation Window` view.

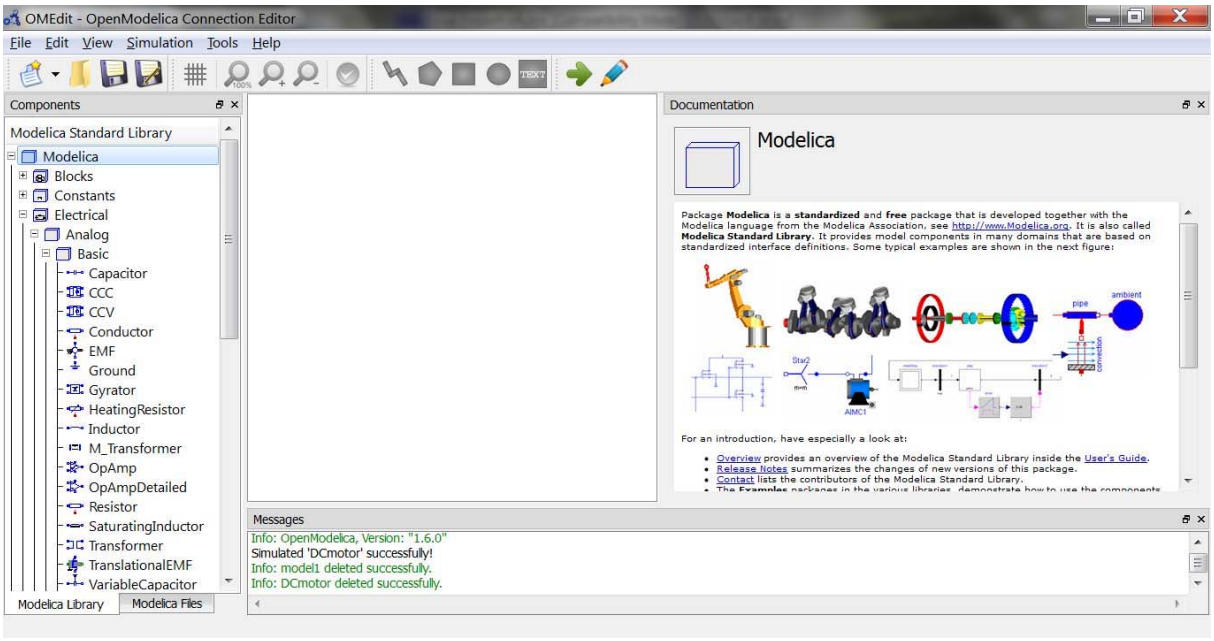


Figure 5-2: Documentation Window

5.2 Dialogs

Dialogs are a kind of sub-windows that are not visible by default. The user has to launch them or they will automatically appear due to some user action.

5.2.1 New Dialog

The `New Dialog` can be launched from `File > New > Model Type`**.** The model type can be `model`, `class`, `connector`, `record`, `function`, or `package`.

5.2.2 Simulation Dialog

The `Simulation Dialog` can be launched either from `Simulation > Simulate` or by clicking on the `Simulate!` button in the toolbar. Figure 4-7 shows a simulation dialog. The simulation dialog allows setting attributes of the simulation. You can set the value of any attribute, depending on the simulation requirement. The simulation attributes are,

- Simulation Interval
  - Start Time
  - Stop Time
- Output Interval
  - Number of Intervals
  - Output Interval
- Integration
  - Method
  - Tolerance
  - Fixed Step Size

5.2.3 Model Properties Dialog

The models that are placed in the `Designer Window` can be modified by changing their properties. In order to launch the `Model Properties Dialog` of a particular model, right click the model and select `Properties`. See Figure 5-3. The properties dialog contains the name of the model, the class name the model belongs to, and the list of parameters of the component.

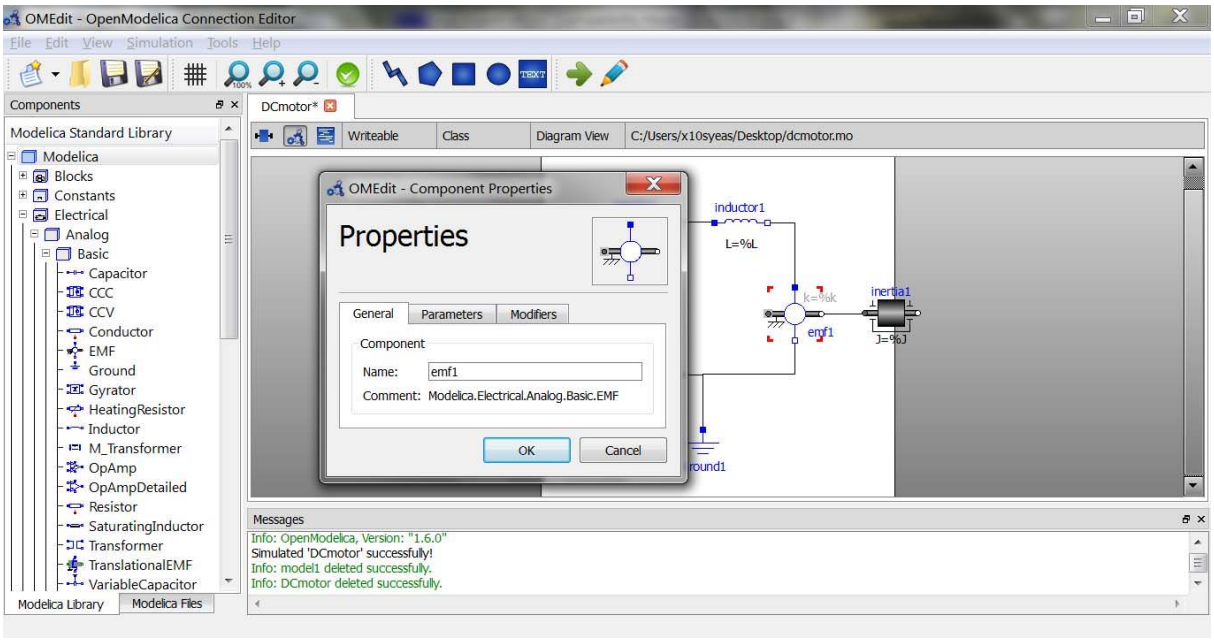


Figure 5-3: Properties Dialog

5.2.4 Model Attributes Dialog

Right click the model placed in the `Designer Window` and select `Attributes`. This will launch the attributes dialog. Figure 5-4 shows the `Model Attributes Dialog`.

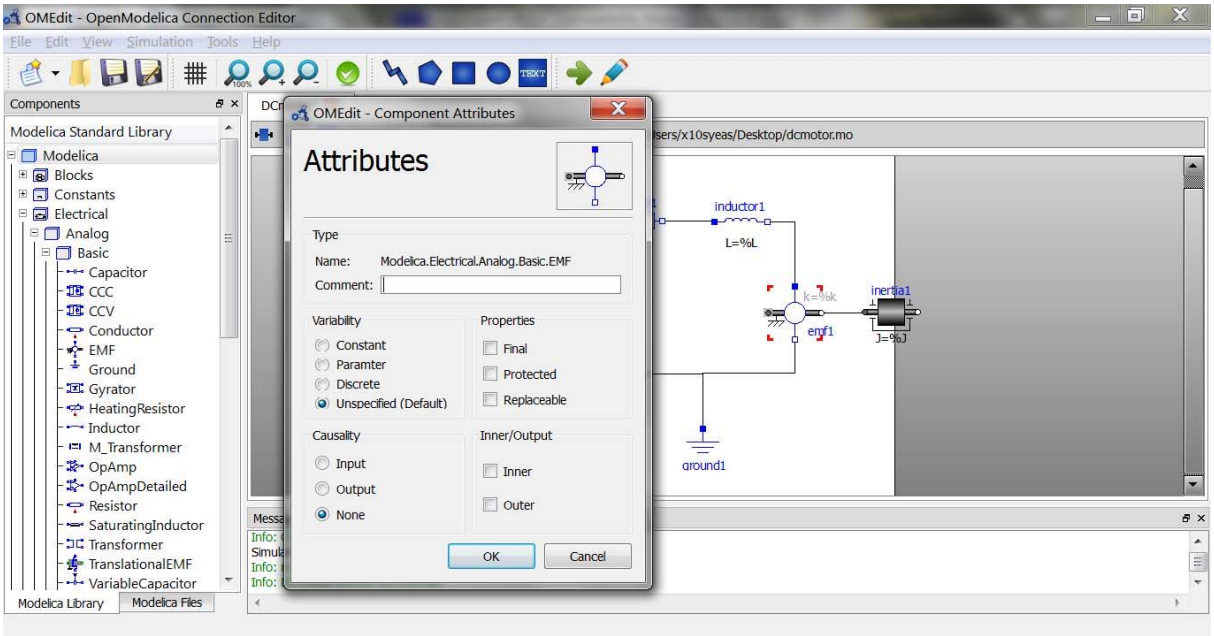


Figure 5-4: Attributes Dialog

Chapter 6 Design and Implementation

- How does the OpenModelica Compiler communication work?
- Improvements done in the OMC API.
- Interpreting Modelica annotations to draw graphical objects.

6.1 Communication with OMC

For graphical modeling OMEdit needs to draw shapes/component models that are defined by Modelica annotations. In order to obtain the Modelica annotations OMEdit must be able to communicate with the OpenModelica Compiler through the Corba interface.

6.1.1 OMC Corba Interface

OMC is the short name for the OpenModelica Compiler. There are two methods to invoke it:

- As a whole program, called at the operating-system level, e.g. as a command.
- As a server, called via a Corba client-server interface from client applications.

OMEdit uses the second method to invoke the OpenModelica Compiler/Interpreter OMC, since this allows interactive access and querying of the models, needed for interactive graphic editing.

6.1.2 The Corba Client Server Architecture

The Figure 6-1 below describes the design of the OpenModelica client server architecture. OMEdit plays the role of client in this architecture. It sends and receives commands through the Corba interface.

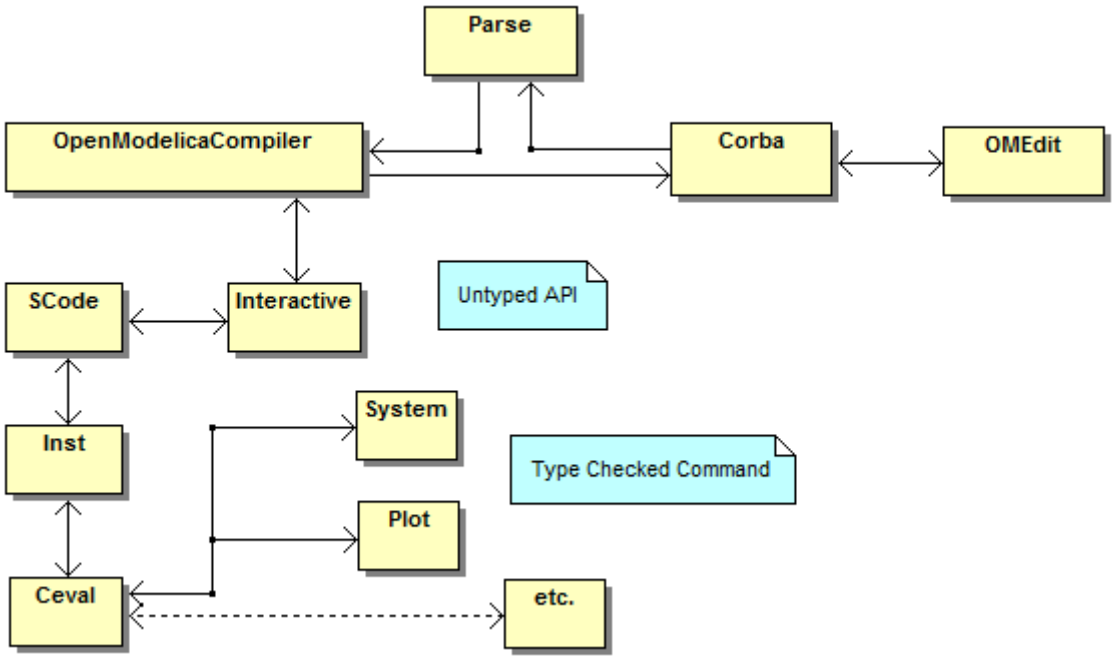


Figure 6-1: Client-Server interconnection structure of the compiler/interpreter main program and some interactive tool interfaces [17]

Messages via 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 consists of 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 the environment [17].

6.1.3 Invoking OMC through Corba

In order to start communication with OMC through Corba we need to start `omc.exe` as a process with special parameters passed to it. The OMC binary executable file is located in `$OPENMODELICAHOME/bin`. OMEdit invokes OMC with the special Corba flag `+d=interactiveCorba` telling OMC to start with the interactive Corba communication environment. The complete command will look like this:

```
omc.exe +d=interactiveCorba.
```

On Linux machines the `omc.exe` is located on the same location without the `.exe` extension. The Corba flags work in the same manner. The complete command on a Linux system will appear as follows:

```
omc +d=interactiveCorba.
```

OMEdit starts a new OMC process for each instance of OMEdit. Only one OMC is linked to each instance of OMEdit. However, for some special tasks a new OMC is used and is removed as soon as the task is completed. For example, this happens for tasks like opening an existing model. The existing model is loaded into OMEdit through the `loadFile` API command. The `loadFile` command loads the model into the OMC global scope, **overwriting any existing model with the same name**. In order to avoid this overwriting behavior of OMC, OMEdit loads the model in a new instance of OMC and then checks whether this model exists in original OMC or not through the `existClass` API command. OMEdit also passes one special argument flag `+c` to OMC which is used to specify the Interoperable Object Reference (IOR) file name. By default the IOR file is created in the temp directory.

```
// create new OMC instance and load the file in it
OMCProxy *omc = new OMCProxy(mpParentMainWindow, false);
QString fileName = path_to_model;
// if error in loading file
if (!omc->loadFile(fileName))
{
    Print_Error_Message();
    return;
}
// get the class names now to check if they are already loaded or not
QStringList existingmodelsList;
QStringList modelsList = omc->getClassNames();
bool existModel = false;
// check if the model already exists in OMEdit original OMC instance
foreach(QString model, modelsList)
{
    if (mpParentMainWindow->mpOMCProxy->existClass(model))
    {
        existingmodelsList.append(model);
        existModel = true;
    }
}
// check if existModel is true
if (existModel)
{
    Print_Error_Message();
    return;
}
else
{
    mpParentMainWindow->mpOMCProxy->loadFile(fileName);
    return;
}
```

OMEdit uses the application session identity number along with the current timestamp to ensure that each instance of OMEdit gets a new OMC. Once OMC is started with the `+d=interactiveCorba` flag, It will create a file named `openmodelica.objid` *(name depends on the*

+c *argument value of OMC)* in the temp directory of the operating system. This file contains the Corba IOR.

6.1.4 What to do with the Corba IOR File?

The IOR file contains the Corba object reference as a string. The Corba object is created by reading the string written in the IOR file. Here is an example with source code for starting OMC and creating a Corba object:

```
QFile objectRefFile (path_to_IOR_File);
int argc = 2;
static const char *argv[] = { "-ORBgiopMaxMsgSize", "10485760" };
CORBA::ORB_var orb = CORBA::ORB_init(argc, (char **)argv);
objectRefFile.open(QIODevice::ReadOnly);
char buf[1024];
objectRefFile.readLine( buf, sizeof(buf) );
QString uri( (const char*)buf );
CORBA::Object_var obj = orb->string_to_object(uri.trimmed().toLatin1());
```

6.1.5 OMC API Enhancements

During the development of OMEdit several issues with the OMC Application Programming Interface (API) were discovered:

- Annotations for some models could not be retrieved via `getIconAnnotation`, `getDiagramAnnotation`, or `getDocumentationAnnotation`.
- `addConnection` and `updateComponent` did not work correctly.
- `renameComponent` was very slow.
- The package `Modelica.UsersGuide` does not have any icon/diagram annotation but instead has a non-standard Dymola annotation.

For example `getIconAnnotation(Modelica.Electrical.Analog.Resistor)` did not work because the `Resistor` model had component references inside the annotations. This problem was solved by symbolically elaborating (instantiating) the `Resistor` model, constant evaluate the `useHeatPort` parameter expression and then elaborating the annotation record with this constant value.

Using constant evaluated parameters from elaborated model does not work for annotations that contain `DynamicSelect` and additional support for such annotations is needed. Unfortunately the `DynamicSelect` annotation creates problems for Modelica software that uses a client-server paradigm as it connects an annotation with a simulation, not with the actual model. However, `DynamicSelect` can still be handled by returning the entire expression to the client who could link a simulation variable to the annotation.

Retrieving the documentation annotation for MSL 3.1 did not work at first because documentation annotations had been moved *(MSL 2.x had no such requirements)* to the end of the class *(typically in an equation section)* and OMC only searches the public declaration sections. This was solved easily in OMC by searching the entire model for the documentation annotation.

To make it easier to find which annotations cannot be retrieved correctly OMC was changed to return the exact annotation that was present in the model. Using this feature the problematic parts of the communication between OMEdit and OMC was debugged.

Updating components and adding connections to classes had small issues in OMC that were fixed to support OMEdit.

The package `Modelica.UsersGuide` and several others do not have any icon/diagram annotation and displaying these packages in the MSL 3.1 browsing tree did not look nice. However, we observed that these packages had a non-standard Dymola specific annotation which is:

`Dymola_DocumentationClass = true`. In order to retrieve this annotation in OMEdit the OMC API had to be extended with a new function: `getNamedAnnotation(Modelica.UsersGuide) => true`. Now these packages can display a predefined icon in the tree browser.

To automatically test which component models had problems a script was written in OMEdit that walks through the entire MSL 3.1 and calls OMC API functions on these models to see if the retrieved information is correct or not. A list with problematic models was built. Subsequently these issues were solved one-by-one.

The function to rename a component, `renameComponent` API function, was extremely slow when MSL 3.1 was loaded. This happened because OMC had to go through all models and components and do a refactoring. We added a new API `renameComponentInClass` that renames the component only in the model that is built using OMEdit and not in any other.

6.2 Annotations

Modelica annotations are used for storing auxiliary information about a model such as graphics, documentation, versioning, etc. [14]. Once OMEdit is connected with OMC it can request the annotations. OMEdit uses three types of annotations;

- Annotations for Graphical Objects
- Annotations for Connections.
- Annotations for Documentation.

6.2.1 Shapes/Component Models Annotations

All the shapes drawn in OMEdit are based on Modelica Annotations standard 3.2. Graphical Annotations consist of two abstraction layers: the icon layer and the diagram layer. The icon layer contains the icon representation of a component and the diagram layer shows the inheritance hierarchy, connections, and inherited component models.

For example, a graphical icon representation of a `Resistor` component will look like this:

```
{-100.0,-
100.0,100.0,100.0,true,0.1,2.0,2.0,{Rectangle(true,{0.0,0.0},0,{0,0
,255},{255,255,255},LinePattern.Solid,FillPattern.Solid,0.25,Border
Pattern.None,{{-70,30},{70,-30}},0),Line(true,{0.0,0.0},0,{{-
90,0},{-
70,0}},{0,0,255},LinePattern.Solid,0.25,{Arrow.None,Arrow.None},3,S
mooth.None),Line(true,{0.0,0.0},0,{{70,0},{90,0}},{0,0,255},LinePat
tern.Solid,0.25,{Arrow.None,Arrow.None},3,Smooth.None),Text(true,{0
.0,0.0},0,{0,0,0},{0,0,0},LinePattern.Solid,FillPattern.None,0.25,{
{-144,-40},{142,-
72}},"R=%R",0,TextAlignment.Center),Line(false,{0.0,0.0},0,{{0,-
100},{0,-
30}},{127,0,0},LinePattern.Dot,0.25,{Arrow.None,Arrow.None},3,Smoot
h.None),Text(true,{0.0,0.0},0,{0,0,255},{0,0,0},LinePattern.Solid,F
illPattern.None,0.25,{{-
152,87},{148,47}},"%name",0,TextAlignment.Center)}}
```

This graphical icon representation of the `Resistor` component is parsed by OMEdit for drawing this component model. The icon annotation is retrieved from OMC through the `getIconAnnotation` API command. Each graphical object is built using the primitive graphical types: `Line`, `Polygon`, `Rectangle`, `Ellipse`, `Text` and `Bitmap`.

The primitive graphical types in OMEdit are handled through the `QGraphicsItem` class of Qt. A `ShapeAnnotation` class was created which is derived from `QGraphicsItem` and `QObject`. This class is an abstract class which contains classes of all primitive graphical elements. See Figure 6-2.

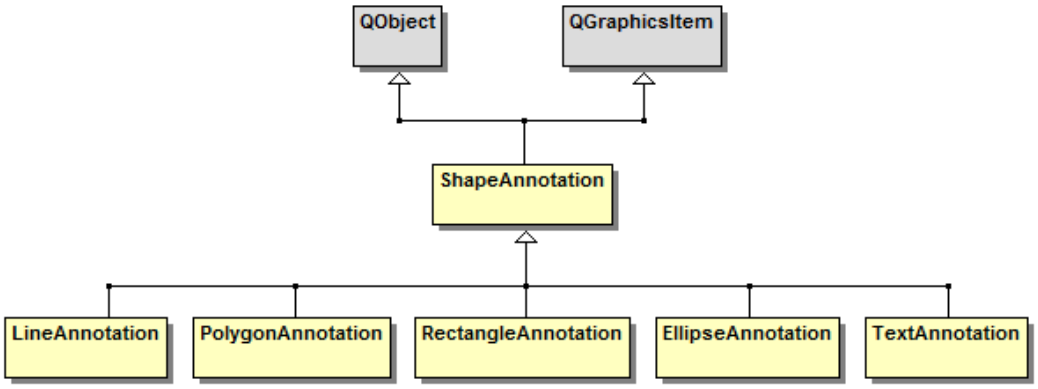


Figure 6-2: Classes hierarchy for predefined graphical elements

6.2.2 Primitive Graphical Types

Each graphical primitive extends from `GraphicItem`. `GraphicItem` is a partial record which defines the model's visibility; origin and rotation.

```
partial record GraphicItem
       Boolean visible = true;
end GraphicItem;
```

`Polygon`, `Rectangle`, `Ellipse` and `Text` also extends from `FilledShape`.

```
record FilledShape "Style attributes for filled shapes"
       Color lineColor = Black "Color of border line";
       Color fillColor = Black "Interior fill color";
       LinePattern pattern = LinePattern.Solid "Border line pattern";
       FillPattern fillPattern = FillPattern.None "Interior fill pattern";
       DrawingUnit lineThickness = 0.25 "Border line thickness"
end Style;
```

The `FilledShape` record is used to define line color, line pattern, fill color, fill pattern and line thickness of a model [15].

6.2.2.1 Line Annotation

A `Line` is a record that extends from `GraphicItem`.

```
record Line
       extends GraphicItem;
       Point[:] points;
       Color color = Black;
       LinePattern pattern = LinePattern.Solid;
       DrawingUnit thickness = 0.25;
       Arrow[2] arrow = {Arrow.None, Arrow.None}; "{start arrow, end arrow}"
       DrawingUnit arrowSize = 3;
       Boolean smooth = false "Spline if true";
end Line;
```

Whenever a `Line` object is found in an annotation string, OMEdit creates an object of `LineAnnotation` which extends from `QGraphicsItem` class of Qt, see Figure 6-2. This class is also used to create a user defined `Line` shape, see Chapter 4, Section 4.4 for detailed information about `Line` shape [15].

6.2.2.2 Polygon Annotation

A Polygon is a closed shape which means its first and last points are always same. However, this is only true if the Polygon contains more than two points. A Polygon extends from GraphicItem and FilledShape,

```
record Polygon
        extends GraphicItem;
        extends FilledShape;
        Point[:] points;
        Boolean smooth = false "Spline outline if true";
end Polygon;
```

OMEdit creates an object of PolygonAnnotation for each Polygon instance [15]. See Figure 6-2.

6.2.2.3 Rectangle Annotation

A Rectangle is created using two points which also acts as the bounding box of the Rectangle. Each Rectangle is represented as an object of RectangleAnnotation class in OMEdit.

```
record Rectangle
        extends GraphicItem;
        extends FilledShape;
        BorderPattern borderPattern = BorderPattern.none;
        Extent extent;
        DrawingUnit radius = 0 "Corner radius";
end Rectangle;
```

The radius attribute defines the roundness of the Rectangle [15]. See Figure 6-2.

6.2.2.4 Ellipse Annotation

An Ellipse is similar to a Rectangle. The only difference is that Ellipse is a rounded shape.

```
record Ellipse
        extends GraphicItem;
        extends FilledShape;
        Extent extent;
end Ellipse;
```

An object of EllipseAnnotation represents the Ellipse instance in OMEdit [15]. See Figure 6-2.

6.2.2.5 Text Annotation

The Text record is defined as follows:

```
record Text
        extends GraphicItem;
        extends FilledShape;
        Extent extent;
        String textString;
        DrawingUnit fontSize;
        String fontName;
        TextStyle[:] textStyle;
end Text;
```

The TextAnnotation class is derived from QGraphicsItem class and uses the textString, fontSize, fontName and textStyle from the Text record to draw a text on the Designer Window [15]. See Figure 6-2.

6.2.2.6 Bitmap Annotation

A BitmapAnnotation class is used to show a bitmap image. OMEdit uses the Bitmap record to render the bitmap,

```
record BitMap
        extends GraphicItem;
```

```
        Extent extent;
        String fileName "Name of bitmap file";
        String imageSource "Pixmap representation of bitmap";
end BitMap;
```

The bitmap can be specified through an external stored file or through the annotations. The bitmap is scaled using the `extent` points given in the record [15].

6.2.3 Connection Annotation

This annotation defines the graphical representation of a connection between two component models. An example of a connection annotation is the following:

```
connect (a.x, b.x)
annotation(Line(points={{-25,30}, {10,30}, {10, -20}, {40,-20}}));
```

A connection annotation is composed of the primitive graphical type `Line`. The points of the line define the connection line co-ordinates between two connecting component models.

OMEdit creates an object of `Connector` class for each connection. Each `Connector` contains instances of `ConnectorLine` depending on the number of points in a connection. The `Connector` class is derived from `QGraphicsWidget` class which is container class for graphical objects. The `ConnectorLine` class is derived from `QGraphicsLineItem` which represents a single line. If we have `n` points in a connection annotation then we have `n-1` instances of `ConnectorLine`. In short `n` number of points creates `n-1` lines. The following Figure 6-3 shows the implementation of connection annotation in OMEdit.

In Figure 6-3 the `GraphicsView` class represents the `Designer Window` since all the connections are drawn in it. `GraphicsView` has a `QVector (mConnectorVector)` which contains all the connections in a model. Each `Connector` has a `QVector (mpLines)` which contains the list of lines that are used to draw a connection between two component models.

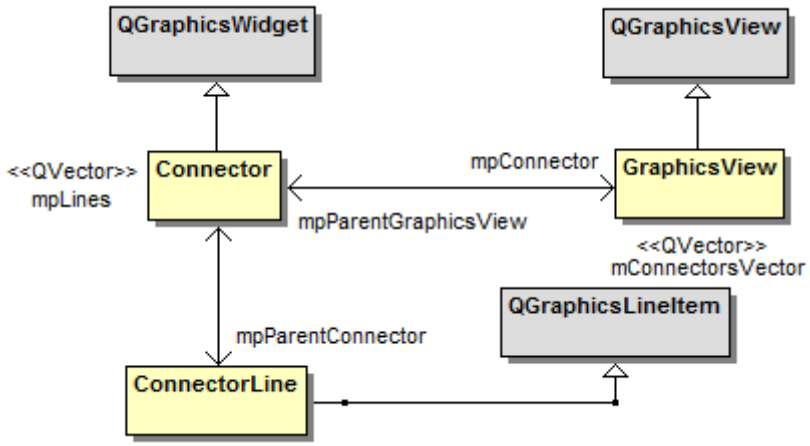


Figure 6-3: Implementation of connection annotation

6.2.4 Documentation Annotation

The Documentation annotation is used for textual descriptions of models. The documentation annotation written as follows:

```
documentation_annotation:
annotation"(" Documentation "(" "info" "=" STRING
["," "revisions" "=" STRING] ")" ")"
```

OMEdit requests OMC for the documentation of a specific component/library through the `getDocumentationAnnotation` command and OMC returns the `info` annotation contained inside documentation annotation which is a string. The tags `<HTML>` and `</HTML>` define the start and end of the string.

The `Documentation Window` is responsible for displaying the retrieved documentation annotation. It uses Qt's Webkit module. The Webkit module is centered on `QWebView` which can display `HTML` content. Since the documentation annotation is an HTML string so we have sub-classed QWebView in a DocumentationViewer class which is contained inside the DocumentationWidget (also called Documentation Window) class. The HTML string of documentation annotation contains four types of links:

- Hyperlinks – Used to navigate to external websites.
- Image Links – Used to reference the local image files.
- Modelica Links – Used for linking to other component models.
- Mailto Links – Used to display email addresses that can be used for future contacts.

`QWebView` has built-in support for images so we didn't have to handle that. We just set the proper base path where all the images were located. However, for hyperlinks and mailto links we used the `QDesktopServices` class. This class uses the default system browser in case of hyperlinks and default email client in case of mailto link. The Modelica links are special links which starts with `Modelica://` and reference to some component model or a package. Consider the following code snippet:

```
// if url contains http or mailto: send it to desktop services
if ((url.toString().startsWith("http")) or (url.toString().startsWith("mailto:")))
{
    QDesktopServices::openUrl(url);
}
// if the user has clicked on some Modelica Links like Modelica://
else if (url.toString().startsWith("Modelica"))
{
    // remove Modelica:// from link
    QString className;
    className = url.toString().mid(10, url.toString().length() - 1);
    // send the new className to DocumentationWidget
    getDocumentationAnnotation(className);
}
```

In the above code snippet the else part handles the Modelica links. Whenever we have a Modelica link clicked we trim the first word of the `URL`, which is `Modelica://`, and then use the remaining part of the `URL` as the argument for `getDocumentationAnnotation` command.

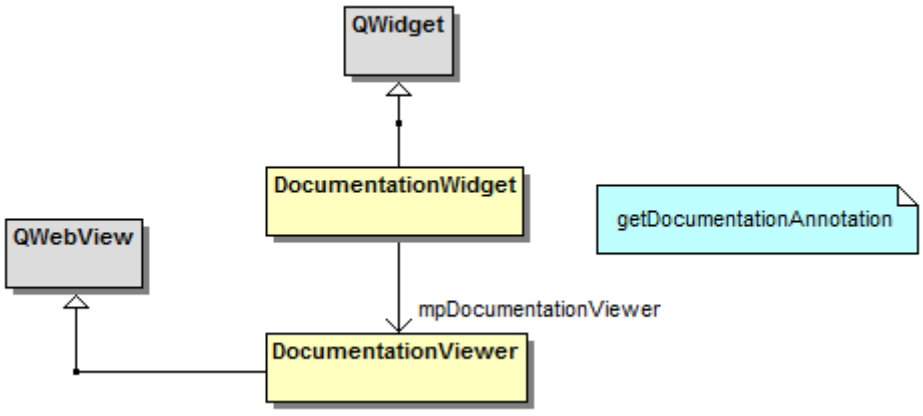


Figure 6-4: Implementation of documentation annotation

6.3 Structure of Classes

The following UML class diagram shown in Figure 6-5 shows the classes hierarchy used in OMEdit.

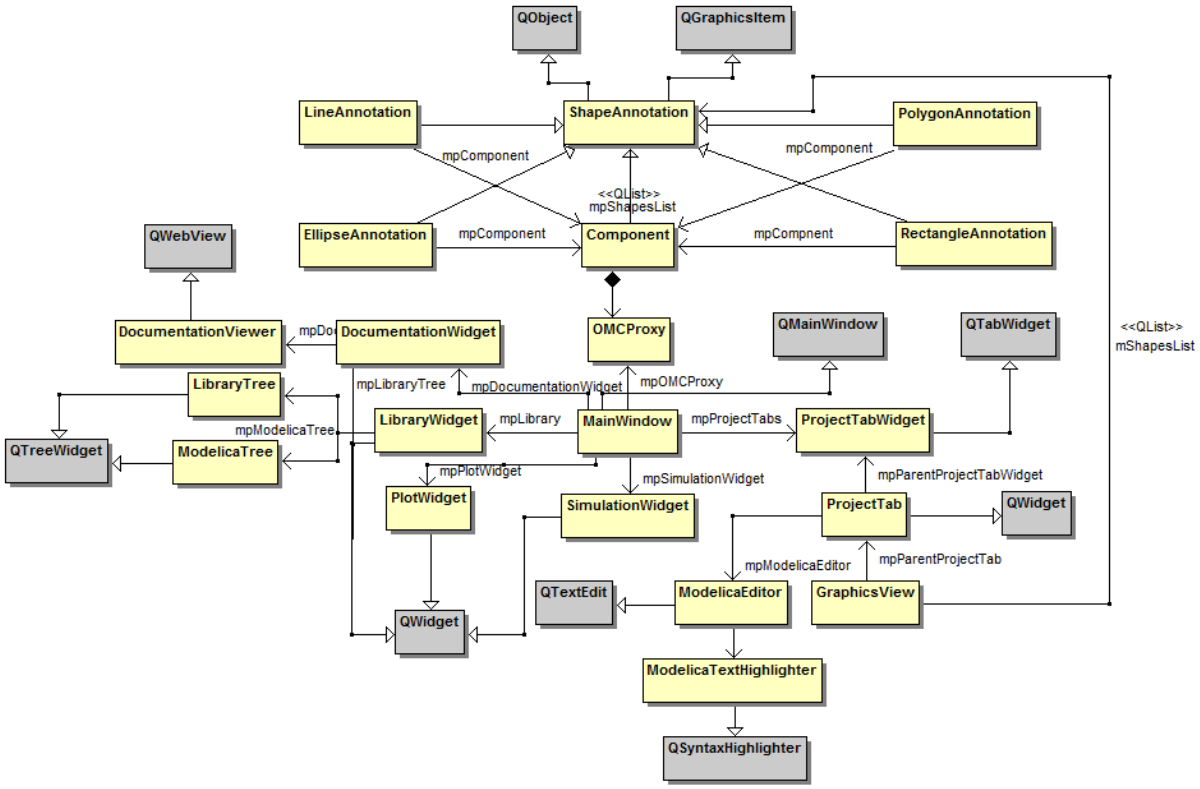


Figure 6-5: OMEdit UML class diagram. The gray shaded classes are Qt core classes. This class diagram is reversed engineered from the source code using BoUML [16]

The Table 6-1 shows the list of classes used in OMEdit. It lists each class with its parent class and also briefly explains the implementation details. Table 6-1 also explains the UML class diagram shown in Figure 6-5.

Class Name	Inherits	Description
Component	ShapeAnnotation	Creates a component model icon using the graphical elements `Line`, `Polygon`, `Rectangle`, `Ellipse`, `Text` and `Bitmap`. Handles all the keyboard and mouse events related to the component model. Also, provides an interface for making connections from the component. This class uses the `Transformation` class in order to transform component model according to the Modelica Annotations.
ComponentsProperties		This class is used to store component model attributes which include name, classname, comment, protected, final, flow, stream, replaceable, variability, inner, outer and causality.
Connector	QGraphicsWidget	For each `connect` statement a `Connector` object is created. This class contains the list of `ConnectorLine` objects where one `ConnectorLine` instance represents one line (one

		Connector can contains multiple lines). See Chapter 6 section 6.2.3.
ConnectorLine	QGraphicsLineItem	This class represents one line in a connection between two component models. All the mouse events are handled in this class like moving and selecting a line.
CornerItem	QGraphicsItem	This represents the angle brackets shown around the component model. This class provides an interface for resizing the component model diagram.
DocumentationWidget	QWidget	Provides a container for showing the documentation for Modelica models and libraries. It contains the object of DocumentationViewer which is used to display Modelica documentation in a web view.
DocumentationViewer	QWebView	The DocumentationViewer class extends from QWebView which provides Qt's built-in support for displaying HTML content. See Chapter 6 section 6.2.4.
EllipseAnnotation	ShapeAnnotation	Draws an ellipse based on Modelica annotations. This class is used to draw an ellipse contained inside the annotations of Modelica's predefined models and also provides interface for drawing user defined ellipse shape.
GraphicsView	QGraphicsView	The GraphicsView class inherits from the QGraphicsView class. QGraphicsView provides a canvas for drawing graphical objects on it. This class is used to show the Modelica component model which is created using the ShapeAnnotation class.
Helper		This class is used for defining error messages, enumerations and text as strings that are used throughout the application.
LibraryWidget	QWidget	This is a container class and contains objects of ModelicaTree and LibraryTree.
LineAnnotation	ShapeAnnotation	Draws a line based on Modelica annotations. It creates a line by reading the annotations of a predefined model and also provides the interface for creating a user defined line shape.
MainWindow	QMainWindow	Represents the whole application interface. This class extends from QMainWindow. QMainWindow class provides a built-in layout which contains menus, toolbars, dock windows, and status bar. OMEdit's all operations are initiated from this class and are then forwarded to the respective class for further handling. For example, if a user presses Ctrl+N, MainWindow handles the user request and then forwards it to ModelWidget which then

		shows the new dialog.
MessageWidget	QTextEdit	This class shows the information, warning, and error messages to the user depending on the operation perform by the user. See Chapter 5 Section 5.1.4.
ModelicaEditor	QTextEdit	Used to display the corresponding Modelica text of the model. This class also provides an interface for searching a string inside the text. The user can launch the search perspective by pressing Ctrl+F.
ModelWidget	QDialog	This class shows a dialog box where the user can enter the name and a new model is created using the name provided by the user. The `MainWindow` class sends the user request to this class. The user can request a new `Model`, `Package`, `Function`, `Block`, `Record`, `Class` or `Connector`. Depending on the user request `ModelWidget` shows the appropriate dialog to the user.
OMCProxy		Provides a platform between the OpenModelica Compiler and OMEdit. All requests to OMC are sent through OMCProxy. This class uses the OmniOrb Corba implementation to communicate with OMC. See Chapter 6 section 6.1.
PlotWidget	QWidget	`PlotWidget` is used to read the simulation result and draws a tree from it. The tree contains the list of instance variables. See Chapter 4 section 4.3.5 and Chapter 5 section 5.1.3.
PolygonAnnotation	ShapeAnnotation	Draws a polygon based on Modelica annotations. It provides an interface for creating a user defined polygon shape and also creates a polygon by reading the annotations of a predefined model.
ProjectTab	QWidget	This is a container class and contains instances of `GraphicsView` and `ModelicaEditor`. Each `ProjectTab` contains two instances of `GraphicsView`, one for the Icon View and the other for the Diagram View.
ProjectTabWidget	QTabWidget	The `ProjectTabWidget` class provides a tab-based container. Each tab of this class represents an instance of `ProjectTab`.
RectangleAnnotation	ShapeAnnotation	Draws a rectangle based on Modelica annotations. It provides an interface for creating a user defined rectangle shape and also creates a rectangle by reading the annotations of a predefined model.
ShapeAnnotation	QGraphicsItem, QObject	`ShapeAnnotation` is the base class for all the shapes used in OMEdit. It handles all the mouse and key board events preformed on shapes. `ShapeAnnotation` inherits from `QGraphicsItem` which makes it a graphical object. It also inherits from `QObject` which allows it to use Qt's Signal

		Slot mechanism.
SimulationWidget	QDialog	This class provides an interface for sending `simulate` commands to OMC. The user starts the simulation process by entering the simulation parameters. Once the simulation is completed this class invokes the `PlotWidget` class which then reads the simulation result.

Table 6-1: OMEdit classes

Chapter 7 Related Work

- SimForge is another open source model editor for OpenModelica.
- Dymola is a dynamic modeling tool and a strong multi-engineering modeling and simulation environment.
- MathModelica is a powerful multi-engineering modeling and simulation environment.

There is already one open source graphical editor available for OpenModelica called SimForge. There are also several commercial tools available for graphical modeling. Those tools are professional products that work well but are not freely available and are not open source.

7.1 SimForge

SimForge is a graphical and textual editor developed by Politecnico di Milano [4] implemented in Java [18]. This editor is not very efficient, i.e., rather slow, from the user interactivity aspect. One of the major reasons behind SimForge slowness is the Java implementation along with the graphical library used Swing [19].

Matthias Kalle Dalheimer [20] compares Java with C++ in three aspects,

- Programmer-efficiency
- Runtime-efficiency
- Memory-efficiency

According to Matthias, Java programmers achieve higher programmer-efficiency than C++ programmers. However, the Java runtime-efficiency is lower than C++. If the programs are CPU bound the Java runtime efficiency becomes even worse. For memory-efficiency Java uses the Garbage Collection mechanism, which is quite remarkable but we have to trade off memory consumption and slower runtime speed. Since, the Java Virtual Machine (JVM) has to keep track of all the allocated memory blocks and their respective references and have to periodically check them and remove any which is not required anymore.

Moreover, a lot of common functionality implemented in SimForge is not very efficient. Few of the problems are listed below:

- In order to make a connection between two component models the user first have to press the shift key and then have to click on the component port to start making the connection.
- If the user moves the two connected models then the connection lines started pointing to some other location instead of the port it is connected to.
- SimForge allows the user to make a connection between two incompatible component models. While OMEdit does not allow connections between two incompatible types.
- When the user drops a component model on the designer view, a pop up will appear asking user to give the name to the component model. However, in OMEdit the unique names are given dynamically and user can change them later on if he/she wants to.
- The support for Modelica documentation in SimForge is very poor. On the other hand, OMEdit provides a very efficient support for component models documentation.

The above mentioned comparison makes a significant difference between SimForge and OMEdit and also depicts that why there is a need for a new graphical model editor for OpenModelica.

7.2 Dymola

Dymola is a product from Dassault Systèmes (DS). Dynamic Modeling Laboratory (Dymola) is a complete tool for modeling and simulation of integrated and complex systems for use within automotive, aerospace, robotics, process and other applications [5].

7.3 MathModelica

MathModelica is a product from MathCore Engineering AB. MathModelica is a powerful, flexible and extensible system for multi-engineering modeling and simulation [6].

Chapter 8 Conclusion and Future Work

- Summarizing the work
- Integration with OMNotebook
- Interactive Simulation
- Integrated OpenModelica Shell

8.1 Conclusion

This thesis has resulted in the development of new graphical connection editor for OpenModelica, OMEdit. In this work, we have tried to overcome the problems that are faced by the OpenModelica users while creating the models in already existing open source editor, SimForge. The focus has been set from the beginning of this project on making OMEdit more user friendly and really easy-to-use, especially for students learning modeling with Modelica.

The new connection editor is using Trolltech's Qt libraries and it supports Modelica Annotations Standard 3.2 when invoking the Modelica Standard Library in the editor. The annotations are fetched through a Corba communication with the OpenModelica Compiler and the editor parses the annotations and draws the component model shapes. It provides easy interface for modeling Modelica models, simulating, and plotting them.

OMEdit uses OMC as a server from where it gets all the necessary information about the Modelica component models. This client-server behavior is also valid for the model simulation carried out through the Corba interface of OMC.

This thesis work started with the promise that the new graphical editor will work on all platforms. We have managed to fulfill this promise to some extent and tested OMEdit successfully on Windows, Linux and Mac OS X.

OMEdit was also designed taking in consideration the compatibility with other OpenModelica clients (i.e., OMNotebook, OMShell). The work on integrating OMEdit with OMNotebook and OMShell is going on and is also discussed in section 8.2.

8.2 Future Work

There is a long list of functions that we wish to realize in the near future. A few of them are listed in the coming sections.

8.2.1 Integration with OMNotebook

OMEdit will provide an environment where connection diagrams can also be stored using the OpenModelica electronic interactive notebook, OMNotebook [21]. The idea is that the user can do the modeling in the connection editor and then can export his (graphical) models to OMNotebook. Moreover, a graphical (and textual) model in a notebook could be loaded into OMEdit for further editing.

A graphic model in an electronic notebook is just an image. The model including its equations, algorithms, annotations etc. could be hidden behind that image. Thus, OMEdit integrated with OMNotebook will allow users to click on the image and launch the model in the connection editor where the user can manage the connections; add/remove component models etc.

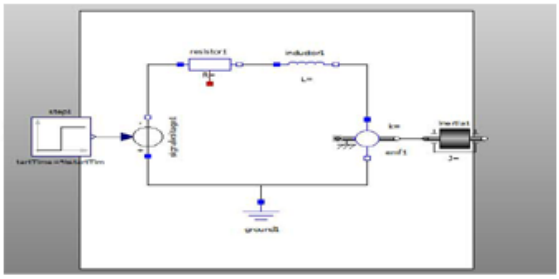
The implementation work for this functionality is ongoing. Figure 8-1 shows an electronic notebook – OMNotebook with `DCmotor` model as an image which is exported from OMEdit. When a user double clicks on the image, the OMEdit editing view is popped up, allowing both textual and graphical editing.

Exercise - Graphical Modeling

1 The DC Motor

A) DC Motor

Make a simple DC-motor using the Modelica standard library that has the following structure:



```
model ...
```

Figure 8-1: OMEdit integrated with OMNotebook

8.2.2 Interactive Simulation

In order to offer a user-interactive and interactive real-time simulation, OpenModelica has an additional subsystem to fulfill general requirements on such simulations, called the OpenModelica Interactive (OMI), shown in Figure 8-2. With OMI the user will be able to simulate the system and interact with it at runtime.

OMI will result in an executable simulation application, such as a non interactive simulation. The executable file will be generated by OMC, which contains the full Modelica model as C/C++ code with all required equations, conditions and different solvers to simulate a whole system or a single system component. This executable file offers both a non-interactive together with an interactive simulation runtime.

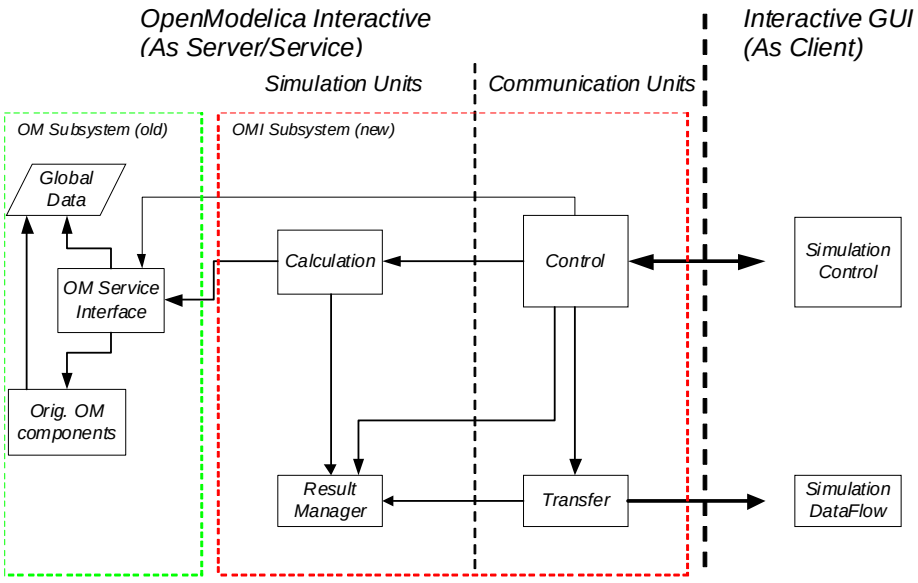


Figure 8-2: OpenModelica Interactive System Architecture Overview [17]

8.2.3 Integrated OpenModelica Shell

The OpenModelica Shell - OMShell is an interactive session handler that parses and interprets commands and Modelica expressions for evaluation, simulation, plotting, etc. The session handler also contains simple history facilities, and completion of file names and certain identifiers in commands [22].

OMEdit will provide an integrated OMShell, which gives users the facility to interact with OMC without loading an external OMShell application.

References

[1] Martin Otter and Hilding Elmqvist. Modelica Language, Libraries, Tools, Workshop and EU-Project RealSim, June 2001. https://www.modelica.org/documents/ModelicaOverview14.pdf, accessed June 2010.

[2] OpenModelica. Modelling and simulation environment for Modelica. http://www.openmodelica.org, accessed December 2010.

[3] PELAB. Programming Environments Laboratory. http://www.ida.liu.se/~pelab/, 4th Decmber 2010

[4] SimForge. Graphical and textual open source model editor, http://trac.ws.dei.polimi.it/simforge/, accessed August 2010.

[5] Dymola. Dynamic modeling tool. http://www.dynasim.se, accessed August 2010.

[6] MathModelica. A powerful, flexible and extensible system for multi-engineering modeling and simulation. http://www.mathcore.com/products/mathmodelica/, accessed August 2010.

[7] IEI. Department of Management and Engineering. http://www.iei.liu.se/?l=en, accessed June 2010.

[8] Mingw. A Minimalist GNU for Windows. http://www.mingw.org/, accessed July 2010.

[9] GCC. The compiler for the GNU operating system. http://gcc.gnu.org/, accessed July 2010.

[10] Doxygen. Generate documentation from source code. http://www.stack.nl/~dimitri/doxygen/, accessed July 2010.

[11] TrollTech, Nokia. Qt a cross platform applicaion framework. http://qt.nokia.com/, accessed Decmeber 2010.

[12] Microsoft. Visual Studio 2010, and integrated development environment. http://www.microsoft.com/visualstudio/sv-se, accessed November 2010.

[13] OmniORB 4.1.4. A robust high performance CORBA ORB for C++. http://omniorb.sourceforge.net, accessed July 2010.

[14] Modelica Association. The Modelica Language Specification Version 3.2, March 24th 2010. https://www.modelica.org/documents/ModelicaSpec32.pdf, accessed November 2010.

[15] Peter Fritzson. Principles of Object-Oriented Modeling and Simulation with Modelica, Wiley-IEEE Press, January 2004.

[16] Bruno Pagès. A free UML tool box. http://bouml.free.fr/, accessed November 2010.

[17] Peter Fritzson, Adrian Pop, Martin Sjölund, Per Östlund, Peter Aronsson. OpenModelica System Documentation Version 1.6, November 2010.

[18] James Gosling. An object oriented programming language developed by Sun Microsystems. http://www.oracle.com/technetwork/java/index.html, accessed September 2010.

[19] Sun Microsystems. Swing a Java GUI widget toolkit. http://java.sun.com/products/jfc/tsc/articles/architecture/, accessed September 2010.

[20] Matthias Kalle Dalheimer. A Comparison of Qt and Java for Large-Scale, Industrial-Strength GUI Development. http://turing.iimas.unam.mx/~elena/PDI-Lic/qt-vs-java-whitepaper.pdf, accessed September 2010.

[21] Anders Fernström, Ingemar Axelsson, Peter Fritzson, Anders Sandholm, Adrian Pop. OMNotebook – Interactive WYSIWYG Book Software for Teaching Programming. In Proc. of the Workshop on Developing Computer Science Education – How Can It Be Done? Linköping University, Dept. Computer & Inf. Science, Linköping, Sweden, March 10, 2006.

[22] Adrian Pop, Peter Fritzson. OpenModelica Shell - OMShell. An Interactive environment for working with the OpenModelica Compiler. http://openmodelica.org/index.php/developer/tools/136, accessed November 2010.

Appendix A – DC Motor Model

The following `DCmotor` model is used to illustrate that how a model is created in OMEdit;

DCmotor

```
model DCMotorCircuit
  Resistor     resistor1(R = 10);
  Inductor     inductor1(L = 0.2);
  Ground       ground1;
  Inertia      inertia1(J = 1);
  EMF          emf1;
  Step         step1;
  SignalVoltage signalVoltage1;
equation
  connect(step1.outPort, signalVoltage1.inPort);
  connect(signalVoltage1.p, resistor1.p);
  connect(resistor1.n, inductor1.p);
  connect(inductor1.n, emf1.p);
  connect(emf1.rotFlange_b, inertia1.rotFlange_a);
  connect(signalVoltage1.n, ground1.p);
  connect(ground1.p, emf1.n);
end DCMotorCircuit;
```

Appendix B – List of OMC API Commands

loadModel(Modelica)	Loads the Modelica Standard Library and stores it in the OMC symbol table. Outputs: true
getIconAnnotation(className)	Get the icon annotation representation of *className* from OMC. Outputs: String
renameComponent(modelName, oldName, newName)	Renames a component model specified inside the *modelName.* Outputs: true
renameComponentInClass(modelName, oldName, newName	Works similar to *renameComponent* command. Only difference is that it only searches the component model inside the *modelName.* Outputs: true
getDocumentationAnnotation(className)	Gets the Modelica documentation of the specified *className.* Outputs: String
getErrorString()	Returns the error string Outputs: String
getClassNames(className)	Returns the list of classes contained inside *className*. The parameter *className* is optional. If not specified OMC will list all the classes that are present in global scope. Outputs: list of classes as string.
getPackages(className)	Works similar to *getClassNames* command but instead of returning all classes it only returns the list of packages. Outputs: list of packages as string.
is*(className)	* indicates *model*, *package*, *class*, *connector*, *record*, *function*, *block* etc. Outputs: true
getClassRestriction(className)	Returns the specific Modelica type of which the particular *className* is. Outputs: *model*, *package*, *class*, *connector*, *record*, *function*, *block* etc.
getParameterNames(className)	Returns the list of parameters of a *className*. Outputs: list of parameters as string.
getConnectionCount(className)	Returns the connections of a specified *className*. Outputs: number of connections as string.
getNthConnection(className, number)	Returns the connection listed at location identified by *number* in a *className*. Outputs: the connection string.
getNthConnectionAnnotation(className, number)	Returns the annotation string of a connection listed at location identified by *number* in a *className*. Outputs: connection annotation string.

getInheritanceCount(className)	Returns the number of inherited instances. Outputs: number of inherited models as string.
getComponents(className)	Returns a list of components in a *className*. Outputs: list as a string.
getComponentAnnotations(className)	Returns the list of component annotations in the same order as components are listed through a *getComponents* command. Outputs: list as a string.
existClass(className)	Checks if the specified *className* exists in the OMC global scope. Outputs: true.
deleteClass(className)	Deletes the specified *className* from the OMC global scope. Outputs: true.