A knowledgebase web platform for collaborative physical system modeling and simulation

Zhang, Yanshan

2012

Zhang, Y. (2012). A knowledgebase web platform for collaborative physical system modeling and simulation. Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/55379>

<https://doi.org/10.32657/10356/55379>

**A KNOWLEDGEBASE WEB PLATFORM FOR COLLABORATIVE
PHYSICAL SYSTEM MODELING AND SIMULATION**

ZHANG YANSHAN

SCHOOL OF MECHANICAL AND AEROSPACE ENGINEERING

2012

**A KNOWLEDGEBASE WEB PLATFORM FOR COLLABORATIVE
PHYSICAL SYSTEM MODELING AND SIMULATION**

ZHANG YANSHAN

School of Mechanical and Aerospace Engineering

A thesis submitted to the Nanyang Technological University in partial fulfillment
of the requirement for the degree of Doctor of Philosophy

2012

Abstract

Engineering study involves concepts, principles and phenomena that are difficult to interpret due to people's cognitive constraints. Learning through experiments is acknowledged to be one of the most effective ways of gaining new knowledge. Computer simulation involves applying scientific theories and performing data analysis on the collected data. Many simulation platforms provide visualization tools for observing phenomena and data analysis, and these are all characteristics of experiments. Hence, computer simulation can be used as a complementary tool to understand concepts, principles and phenomena other than actual physical experiments.

Today's Internet has the ability and capacity to share and publish large quantities of digital content. The Internet is becoming the most common channel for learning and sharing knowledge. The most popular places where people learn and share knowledge on the Internet is World Wide Web (WWW). However, there are not many simulation tools that support simulation capability and experiments sharing capability over the Web. In addition, traditional web document types such as text, image, and video are not able to describe abstract terms that are usually expressed or demonstrated using simulation experiments. A web-based platform, called Proteus (<http://www.visualphysics.net/pweb>), has been developed to provide a place where students and educators can easily create and share their computer models of physical systems. It comes with a web-based modeling and simulation tool called ProteusGWT (<http://www.visualphysics.net/ProteusGWT>). ProteusGWT uses

an intuitive component-oriented approach to the modeling of physical systems spanning multiple domains including systems containing mechanical, hydraulic, thermal, control, electrical, electronic, electric power or process-oriented subcomponents. It synthesizes state-of-the-art web technologies, computational methods for physical systems modeling and simulation to create a computing environment that is widely deployable and scalable. Anyone with a computer or browser-enabled device will be able to use it. Hence, anyone can contribute to this platform. As the platform grows, it could turn out to be a reference for all kinds of physical-system models on the Internet.

Together with this platform, a new web document format, named ModelicaJSON that can represent and allow editing of simulation models. Currently, there is no standard web document format for simulation experiments. Defining the structure of this new web document format that can be used as a web standard is critical. Users have been looking for proper simulation languages to better describe complex composite physical systems. Modelica is a language specially developed for physical system modeling and simulation with numerous advanced features such as object-oriented design, equation-based and a-causal modeling, multiple inheritances and algorithms. It has been developed in an international effort with the aim of creating a unified modeling language for complex physical systems spanning multi-domains. It is an open standard. There are many free and commercial Modelica libraries that are developed with it. Hence, Modelica has been selected as a reference for the development of this new web document format. The new web document format which can represent simulation models and enable them to be edited

via the web-based graphical tool-ProteusGWT also implements key modeling features of the Modelica language.

The main contributions of this research is the development of a knowledgebase web platform for physical modeling and simulation models. The web platform acts as a repository for simulation models to facilitate better sharing and collaboration. The thesis also proposes an intuitive and interactive way of presenting information on the web by providing a web-based, easy-to-use, easily accessible modeling and simulation tool called ProteusGWT. It is the first web-based modeling and simulation tool capable of modeling and simulating multi-domain physical systems that can also include mechanical, electrical, fluid, heat, magnetic and control elements. It proposes a complete implementation of a Modelica parser which can be extended to become a full featured Modelica compiler in future. Another contribution of this thesis is the new web document format that can be used to describe simulation models of physical systems. It puts forward a new standard format for representing a particular type of information on the web, i.e., information on the composition and inner workings of a machine, a mechanism, or a physical phenomenon. Based on the structure of the new web document format, a Modelica MySQL database of Modelica standard library 3.1 is produced to allow easy searching and data manipulation and which can be used for future research. The thesis also includes a detailed review of existing literature on web-based modeling and simulation.

Acknowledgements

I would like to express my sincere gratitude to all the people who have helped and encouraged me during my PhD study at Nanyang Technology University, especially in the process of writing this thesis.

First of all, I would like to thank my supervisor, Prof. Yap Fook Fah, for his continuous encouragement, insightful comments and inspirations. I am deeply indebted to him for his invaluable help and support along the way.

Many thanks to my teammates at ADAM (Advanced Design and Modeling Laboratory), Jasleen Singh, Lei Ting, Gao Lei, and Maryam Khademi, for many rewarding discussions and for contributing to the Proteus-system.

Thanks to all my friends, technicians in ADAM and MOM (Mechanics of Machines Laboratory), for creating a warm working environment. This work would not have been possible without their support.

Thanks to all the people in the Modelica community who are actively involved in the development and maintenance of Modelica Standard Library and Modelica Specifications.

Thanks to my parents and my younger sister for their love and support in many aspects of my life.

Last but not least, I would like to express my deepest love to my girlfriend, Zhang Wanqiao for her understanding and care during the time I was writing this thesis.

Dedicated to

My Parents

Contents

Abstract	i
Acknowledgements	iv
Contents	vi
List of Figures	x
List of Tables	xiii
Chapter 1 Introduction	1
1.1 Overview	1
1.2 Motivation	2
1.3 Research Scope and Objectives	3
1.4 Contributions of Thesis	5
1.5 Thesis Structure	7
Chapter 2 Literature Review	9
2.1 Overview	9
2.2 Outline	10
2.3 Web-based Simulation	10
2.3.1 Client-side Simulation and Visualization	12
2.3.2 Server-side Simulation and Visualization	13
2.3.3 Hybrid Simulation and Visualization	14
2.4 Web-based Simulation Technologies Overview	15
2.4.1 Middleware of Web-based Simulation	15
2.4.2 Front-end Visualization Technologies	20
2.4.3 Rich Internet Application	23
2.4.4 Database Technologies	25
2.5 Modeling and Simulation Language and Tools Overview	27
2.5.1 Use Programming Language for Simulation	28
2.5.2 Emerging of Simulation Language	29
2.5.3 Simulation language for Multi-Domain Physical Systems	30
2.5.4 Modelica Language	33
2.5.5 Other Languages and Tools	36
2.5.6 Functional Mock-up Interface	39

2.6 Web-based Simulation Tools and Other Design Considerations	40
2.6.1 Object-Oriented Vs. Process-Oriented Modeling	41
2.6.2 Model Repository for Storing and Exchanging models	45
2.6.3 Compiler Design of Modelica	48
2.6.4 Hardware-In-Loop (HIL) Modeling and Simulation	53
2.7 Web Application Deployment	56
2.7.1 Limited Computing Resources Issues	56
2.7.2 Performance and Scalability Issues.....	59
2.7.3 Browser Compatibility Issue.....	63
2.8 Conclusion.....	64
Chapter 3 A Knowledgebase Web Platform for Physical System Modeling and Simulation	66
3.1 Introduction	66
3.2 A Knowledgebase Platform for Physical Systems.....	67
3.2 Simulation Experiments and Internet Sharing	68
3.3 Information Representation for Playing.....	72
3.4 Proposed Solution	75
Chapter 4 Design of Proteus - the Web Platform for Physical Systems Modeling	78
4.1 System Overview	79
4.2 A Scenario of the Proteus System in Use.....	81
4.3 Software Structure and Subsystems of the Proteus System	85
4.3.1 Search Engine on the Proteus System	86
4.3.2 File Upload System.....	89
4.3.3 Comment and Rating System.....	91
4.3.4 Category and Tag System.....	92
4.3.5 Other Subsystems	93
4.4 The Contributions.....	94
4.5 Conclusion.....	94
Chapter 5 Design of the Web-based Modeling and Simulation Tool	96
5.1 Overview of the Structure of ProteusGWT	97
5.1.1 Software Design Pattern	97
5.1.2 Appearance of the Web Application.....	101
5.1.3 Internal Structure of ProteusGWT	103
5.2 ProteusGWT Toolbar with Standard Functions.....	105

5.3 ProteusGWT Library Visualization Module	106
5.3.1 Structure of the Library Panel	106
5.3.2 Library Pre-Process Engine	108
5.3.3 ProteusGWT Library Icon Loading Algorithm.....	110
5.4 The Myth of the Graphic Drag-and-Drop Canvas	112
5.4.1 Annotation Structure of a Modelica Model	113
5.4.2 Drawing in ProteusGWT	116
5.4.3 Coordinate system transformation	118
5.4.4 Flip and Rotation Manipulation on Canvas	121
5.4.5 Response time optimization on the canvas	122
5.4.6 Connector Look-up	124
5.5 The Code Generator - Another Use of Modelica Parser	127
5.6 The Simulation Process of ProteusGWT	127
5.7 Feedback - The Message Center	128
5.8 Visualization of Simulation Result.....	129
5.8.1 2D Visualization.....	130
5.8.2 3D Visualization.....	131
5.9 Applications.....	133
5.10 Conclusion.....	139
Chapter 6 Design of the Modelica Parser and ModelicaJSON Standard	140
6.1 Introduction	140
6.2 The Overall Architecture	141
6.2.1 The Big Picture.....	141
6.2.2 A Modelica Parser	142
6.2.3 An Example of using Modelica Parser.....	147
6.3 The Lexical Analyzer and Parser	149
6.3.1 Modelica Lexical Analyzer.....	150
6.3.2 Modelica Grammar	150
6.3.3 Modelica Tree Structure	152
6.4 Modelica Semantics.....	155
6.4.1 Scoping and Name Look-up	155
6.4.2 The Basic Name Look-up Algorithm.....	157
6.4.3 Modelica Semantics and Algorithms	159

6.5 ModelicaJSON Standard Proposition	162
6.5.1 The Web Friendly Model Structure	162
6.5.2 Modelica Database.....	173
6.6 Conclusion.....	174
Chapter 7 Use of Proteus web Platform for Hardware-In-the-Loop Simulation	176
7.1 Introduction	177
7.2 Simulation Experiment Setup	181
7.3 Simulation Result	183
7.4 Issues and Problems	184
7.5 Conclusion.....	186
Chapter 8 Conclusions and Future Work	187
8.1 Summary	187
8.2 Future Work	190
Reference.....	192
Appendix A.....	201
Appendix B.....	204
Appendix C.....	211

List of Figures

Figure 2.1. The Outline of Chapter 2	10
Figure 2.2. A Basic Client-side Simulation and Visualization Configuration	13
Figure 2.3. A Server-side Simulation and Visualization Configuration.....	14
Figure 2.4. A Hybrid Simulation and Visualization Configuration.....	15
Figure 2.5. Comparing Synchronous and Asynchronous Communication	16
Figure 2.6. A classification of Middleware Technologies.....	17
Figure 2.7. “Lego-like” Programming for Kids	42
Figure 2.8 Object-Oriented Modeling vs. Process-Oriented Modeling (This picture is from the Wolfram SystemModeler Website).....	44
Figure 2.9. Typical stages of Translating and Compiling a Modelica Model	49
Figure 2.10 The definition of Key Lexical Units of Modelica Language (As appeared in Modelica Specification 3.1).....	50
Figure 2.11. A Typical 3-tier Web Application Server Architecture	61
Figure 2.12. A Master and Slave Database Structure.....	63
Figure 3.1. The Animated Double Pendulum Model in Matlab File Exchange	73
Figure 3.2. The Double Pendulum Physical System in Wolfram Alpha	74
Figure 4.1. The Physical System and the Computer Model of a Mass-Spring System.....	78
Figure 4.2. The System Architecture of Proteus	80
Figure 4.3. The Main Interface of the Proteus Web Platform	81
Figure 4.4. A Scenario of the Proteus System in Use.....	83
Figure 4.5. A Double Pendulum Physical System.....	84
Figure 4.6. Trying out the Experiment in the Web-based Simulation Tool	84
Figure 4.7 Playing with the Double Pendulum Model in the Web-based Simulation Tool	85
Figure 4.8. Proteus Software Architecture	86
Figure 4.9. The Web Interface of the Proteus Upload Module	90
Figure 4.10. The Tag Cloud Module in the Proteus System	93
Figure 5.1. Decoupling of a Web Application.....	98
Figure 5.2. Overview of the ProteusGWT Platform.....	102
Figure 5.3. ProteusGWT Dialogs and Windows: New Model Dialog, Set Component Attribute Dialog, Set Component Parameter Dialog, Simulation Dialog, Simulation Result Plot Option Dialog, and Simulation Result Visualization Window.....	103

Figure 5.4. Overall Modularized Structure of the ProteusGWT Front-end.....	104
Figure 5.5. Modularized structure of ProteusGWT back-end	105
Figure 5.6. Proteus Library Composition	106
Figure 5.7. A Searchable Component-based library.....	107
Figure 5.8. The Document View of a Resistor Block	108
Figure 5.9. MSL 3.1 Database.....	110
Figure 5.10. The On-demand Image-loading Algorithm.....	111
Figure 5.11. A Description of Annotation (From Modelica Specification 3.1)	114
Figure 5.12. The Annotation Section of a Resistor Component.....	115
Figure 5.13. Modelica Line Annotation Definition.....	116
Figure 5.14. UML Class Diagram of the Graphic Drawing in ProteusGWT	117
Figure 5.15. An Example of the Modelica Resistor class which contains two small classes: A Positive Pin and a Negative Pin.....	120
Figure 5.16. The Flip and Rotation State Machine for ProteusGWT Graphic Blocks	122
Figure 5.17. A Modelica Resistor Example for Connectors' Illustration.....	126
Figure 5.18. ProteusGWT Simulation Flow	128
Figure 5.19. A Sample of the Simulation Result of a Variable	130
Figure 5.20. An example of using the Multibody Visualizer Package	132
Figure 5.21. A Two-mass Two-spring System.....	133
Figure 5.22. The Complete Model of the Two-mass Two-spring System in ProteusGWT	134
Figure 5.23. Displacement for m_1 and m_2 in Plotter Module.....	135
Figure 5.24. The Modelica model of an RC circuit.....	136
Figure 5.25. Voltage Plot of Simulation Result for the Resistor and Capacitor.....	137
Figure 5.26. An Example of Mixed Domain Modeling (DCMotor)	138
Figure 5.27. Angular Velocity of Inertia	138
Figure 6.1. A Modelica Parser to Generate a Web Friendly JSON File.....	144
Figure 6.2. A text expressed in both JSON and XML Format	146
Figure 6.3. The Modelica Source Code of a Resistor.....	148
Figure 6.4. The Graphic Representation of a Resistor Component in ProteusGWT.....	149
Figure 6.5. A snapshot of a Modelica Grammar Specification	152
Figure 6.6. The Abstract View of a Modelica Model.....	153
Figure 6.7. A snapshot of the Proteus Modelica Parser.....	154
Figure 6.8. The Tree Structure of Modelica Language	157
Figure 6.9. A ModelicaJSON Generator	162

Figure 6.10. A Web Friendly Document Structure - ModelicaJSON.....	164
Figure 6.11. The Detailed Structure of a ModelicaJSON Model	165
Figure 6.12 The Structure of Modelica Pre-defined Types	169
Figure 6.13 The common kind of Modification for IncludeComponent and ExtendComponent	172
Figure 6.14. Modelica DB Structure	174
Figure 7.1. An Emant Bluetooth DAQ light sensor kit with a Bluetooth Dongle.....	179
Figure 7.2. A Real-Time HIL Simulation Setup	179
Figure 7.3. A Conceptual View of Real-Time HIL Simulation System.....	181
Figure 7.4. The Schedule Diagram of Real-time HIL simulation	183
Figure 7.5. The Simulation Result of a Real-time HIL Simulation.....	183

List of Tables

Table 6.1. Attributes of Component Element.....	166
Table 6.2. Attributes of IncludeComponent Element.....	167
Table 6.3. Attributes of Connection Element.....	168
Table 6.4. Attributes of Parameter Element	169
Table 6.5. Attributes of Pre-defined Modelica Types	171

Chapter 1

Introduction

1.1 Overview

Web-based applications are readily accessible, low-cost, easy to use and can be delivered quickly and seamlessly. As internet connectivity becomes more prevalent and pervasive and back-end infrastructure more robust and secure, organizations are migrating both data and applications online. In the past few years, as summarized in Taivalsaari's paper [1], many promising web technologies such as Ajax, Adobe Integrated Runtime (AIR), Google Web Toolkit (GWT), Google Gears, JavaFx, Java Applet, Microsoft Silver Light and Ruby on Rails have been created for developing web-based applications. The web browser has in itself become a platform for running applications, which is something that had never originally been intended, whilst web-based applications development has become the future for software development. The emergence of cloud computing has resulted in the availability of more robust and reliable server technologies and this, in turn, has influenced and encouraged the migrating process. The key characteristics of cloud computing services are ease of use, low cost, platform and location independence, reliability, and scalability. The benefits offered by these features will support a natural transition of today's applications into the cloud.

1.2 Motivation

Many undergraduate engineering courses involve concepts, principles and phenomena with which students are unfamiliar. Whilst it is difficult to explain them using traditional presentation methods such as text and visual aids, they can more easily be demonstrated and clarified using experiments in the laboratory. To build a solid foundation of knowledge and a deep understanding it is crucial to experiment, to test certain concepts or determine how a principle works in different situations and circumstances. Learning through experimentation is acknowledged to be one of the most effective ways of gaining new knowledge. Computer simulation involves applying scientific theories and performing data analysis on the collected data. Many simulation platforms provide visualization tools for observing phenomena and for data analysis and these are all characteristics of experimentation. In this way, computer simulation can be seen as another means of experimentation. However, there are few simulation tools currently available to facilitate this kind of active exploration. Existing tools such as Working Model, Visual Nastran, Adams, Cosmos, Dymola, and SimulationX offer simulation but also have several drawbacks such as the need to install software on the local computer. They also involve the user in a steep learning curve and are expensive.

Today's Internet has the ability and capacity to share and publish large quantities of digital content. The Internet is becoming the most common channel for learning and sharing knowledge. The most popular places where people learn and share knowledge on the Internet is World Wide Web (WWW). However, there are not many simulation tools that

support simulation capability and experiments sharing capability over the Web. In addition, traditional web document types such as text, image, and video are not able to describe abstract terms that are usually expressed or demonstrated using simulation experiments.

The emergence of cloud computing has enabled software vendors to offer low-cost and convenient web services that provide essentially similar functionality to the more expensive software which also needs to be installed onto a local computer. It would be useful for users from all over the world to have a web-based open platform in which to explore the principles of physical systems by allowing them to create and simulate simulation models anywhere, any time and free-of-charge. Moreover, the sharing capability of the Internet should allow users to share ideas and views as well as comment on each other's simulation experiments or to perform collaborative experiments online using each other's available resources. However, by the time this thesis was written, nothing quite like this is currently available on the web.

1.3 Research Scope and Objectives

The main objective of this thesis is to develop a knowledgebase web platform for users from all over the world to explore the principles of physical systems by allowing users to create and simulate simulation models anywhere, any time and free-of-charge. Moreover, this platform should allow users to share ideas and views as well as comment on each other's simulation experiments or to perform collaborative experiments online using each other's available resources.

However, traditional web document types such as text, image, and video are not able to describe abstract terms that are usually expressed or demonstrated using simulation experiments. Over the years, many simulation languages with distinct modeling features have been designed with the express purpose of describing complex simulation experiments, for example, object-oriented modeling, multi-domain modeling and a-causal modeling features. However, documents described with these languages cannot be used directly on the web. Currently, there is no standard web document format for simulation experiments of physical systems. Hence, defining the structure of a new web document format that can be used as a web standard is an important objective of this study.

Once the new web document format has been defined, the way it is expressed on a browser will directly affect the end-user experience. For example, graphics (either 2D or 3D) is more intuitive than text, object-oriented blocks are more meaningful than process-oriented blocks, and actively playing with a visual object is more intuitive than reading, listening, and watching. Hence, a sub-objective is to design a more intuitive and interactive representation of the physical system on the web.

The new web document requires a web-based tool to provide display and edit facilities within a web browser. There are still many problems concerning building a web-based tool that researchers need to address. For example, web browsers have limited resources and are consequently unable to deal with web-based tools that are too heavy in terms of startup

time, memory usage and power consumption. Web-based tools are expected to have the same operation response time as desktop applications so the critical logic must be stored and executed locally within a browser. Web-based tools are expected to be used simultaneously by a large number of users so there are also scalability issues that need to be addressed. Hence, another sub-objective is to design a web-based simulation tool which has the same operation response time as desktop applications to provide display and edit facilities within a web browser.

Some simulation experiments often involve hardware interactions such as hardware-in-the-loop (HIL) simulation. In order to make the proposed web platform becomes more open and can be used by users from all over the world to explore the principles of physical systems. For example, researchers from different physical locations can share hardware to run hardware-in-the loop simulation over the web platform. Alternatively, students who want to continue to explore ideas after school can use this method to control laboratory equipment remotely. Thus, another sub-objective is to explore a possible way of perform and share hardware-in-the-loop simulation experiments on the knowledgebase web platform.

1.4 Contributions of Thesis

The main contributions of this research are listed below.

1. The main contribution of this thesis is a knowledgebase web platform for users from all over the world to explore the principles of physical systems by allowing users to create and simulate simulation models anywhere, any time and free-of-charge. Moreover, the web platform also acts as a repository for simulation models to facilitate better sharing and collaboration. It is the first ever online physical model repository (Chapter 4). To carry out this, it is required to dive in many disciplines and topics, as multi-domain physics modeling and simulation, programming in different languages, distributed computing and communication, graphics, parsers development, databases, etc. Research issues such as quality control of the published contents, repository search engine indexing have been investigated and discussed.
2. This thesis proposes a new web document format that can be used to describe simulation models of physical systems. It puts forward a new standard format for representing a particular type of information on the web, i.e., information on the composition and inner workings of a machine, a mechanism, or a physical phenomenon. A detailed description of this new web document format is given in Chapter 6.
3. The thesis also proposes an intuitive and interactive way of presenting information on the web by providing a web-based, easy-to-use, easily accessible modeling and simulation tool called ProteusGWT. It is the first web-based modeling and simulation tool capable of modeling and simulating multi-domain physical systems that can also include mechanical, electrical, fluid, heat, magnetic and control

elements (Chapter 5). Issues such as limited browser resources, fast start-up and responsiveness of the application have been explained and justified.

4. It proposes a complete implementation of a Modelica parser which can be extended to become a full featured Modelica compiler in future (Chapter 6).
5. Based on the structure of the new web document format, a Modelica MySQL database of Modelica standard library 3.1 is produced to allow easy searching and data manipulation and which can be used for future research (Chapter 6).
6. The thesis also includes a detailed review of existing literature on web-based modeling and simulation (Chapter 2).
7. This thesis also presents a way of performing hardware-in-the-loop simulation experiments on the proposed web platform (Chapter 7). Research issues such as time synchronization in hardware-in-the-loop simulation have been discussed.

1.5 Thesis Structure

The thesis consists of eight chapters.

Chapter 2 reviews the relevant literature addressing the issues of web-based modeling and simulation including web technologies, modeling languages and tools. The review also covers research that deals with the performance and scalability issues of a web application.

In Chapter 3, the research problems concerning web-based simulation are fully discussed and a web-based knowledge base for physical systems modeling and simulation is proposed.

Chapter 4 outlines the design and development issues of the knowledgebase web platform and summarizes its main contributions. This platform comes with a web-based modeling and simulation tool which is presented separately in Chapter 5.

The particular challenges of this modeling and simulation tool are presented in Chapter 5 including structure design, graphic drawing, library (i.e., Modelica) processing, and a large number of image loading. At the end of this chapter, several applications of the use of this tool are also presented.

Chapter 6 touches on the design, principles, and rules of a web document format, called ModelicaJSON. Additionally, a parser engine for processing modeling libraries and generating Modelica code is presented.

Chapter 7 presents some other studies in the use of the proposed web platform for hardware-in-the-loop simulation.

Finally, the conclusions of this research are drawn in Chapter 8 and the opportunities for further work are also summarized.

Chapter 2

Literature Review

2.1 Overview

As the Internet has developed, more and more content, including educational material has been created and published on the web. Research [2] shows that in July 2012 the number of web servers had increased to 665 million, 4 times the number in 2009.

The significant increase in the amount of published web content has been accompanied by a number of changes to the ways in which it can be presented online. It is now possible, for instance, to present simulation models on the web. Traditionally, the rich elements used to present web information have consisted of text, image, video, and audio. Some common formats include pdf for documents, jpg for photos, flv and mp4 for video, and mp3 for audio. These standard formats allow for easy publishing and sharing online, and there are various viewer/reader software programs readily available to facilitate this. Traditional simulation programs, however, have remained on the desktop, primarily because there are few web-based simulation viewer/reader software programs available to enable effective online use. Fortunately, web technology has now evolved sufficiently to enable the creation of a simulation viewer/reader that can use simulation models as a new web document. This chapter reviews and discusses the related literature.

2.2 Outline

This chapter provides an overview of different web-based simulation architectures, as shown in Figure 2.1, followed by a survey on related web technologies, and then a review of modeling and simulation technologies. At the end of the chapter, the deployment of a web application is presented and discussed.

Figure 2.1. The Outline of Chapter 2

2.3 Web-based Simulation

Web-based simulation (WBS) utilizes both web and simulation technologies to provide a simulation service on the web which are readily accessible, low-cost, easy to use and can

be delivered quickly and seamlessly. The client computer does not need to install any software as everything should happen within a web browser. WBS enables a closer collaboration and easier sharing over the Internet. The Internet and World Wide Web were developed by Tim Berners-Lee of CERN (Centre for European Nuclear Research) in the early 1990s [3]. This medium comes with rich multimedia features such as hyperlinks, text, images, audio, and video which enable immense quantities and types of content to be published digitally on the web. In future it should also be possible to make simulation available for digital publication [4] using a new web document format.

Work on web-based simulation began in 1995 [3] with the provision of web-front ends to simulations running as Common Gateway Interface (CGI) scripts or programs. The development of Java as a simulation development language (e.g., simulation package or toolset) has provided a major step forward in this technological field and has attracted significant attention from WBS researchers. Of critical importance is that a Java-based simulation tool can be accessed from anywhere on the web. An article by Fishwick [5] presented at the 1996 Winter Simulation Conference was the first research study to discuss WBS as a real possibility. WBS then became a popular area of research and at the 1998 Winter Simulation Conference just two years later several WBS research papers were presented. Much of the research in these papers focused on existing technologies such as Java, Java Applet, HTML, CGI, and CORBA.

In terms of classifying WBS, different researchers [3, 6, 7] have used different classifications. Soliman and Alfantookh [4] give a 3-category classification of WBS systems depending on where the simulation is executed and the visualization generated: client-side simulation and visualization, server-side simulation and visualization, and hybrid simulation and visualization. Some WBS systems fall outside these classifications but the differences can be considered simply as additions and variations. Some WBS systems for example are more interactive than others, and some WBS systems focus on a central repository for model sharing and reusing. More information about these classifications is given below.

2.3.1 Client-side Simulation and Visualization

The terms client and server originate in distributed systems technology. With WBS, a web browser or the web application running inside the browser is the client, and the server could be either a single web server or a group of servers depending on the precise configuration. Client-side simulation and visualization, as you might expect, is a term used to describe simulation and visualization activity that takes place on the client. This method of delivering simulation and visualization is achieved using Java. The emergence of Java Applet has been crucial in enabling a piece of code to be sent to and executed on a client's browser. This, in turn, has led to the development of a client rich enough in features to allow modeling, simulation and visualization to be executed within a browser. Java is available on a variety of different platforms and can tap into numerous third-party libraries. This makes Java the preferred option for developing WBS applications. As summarized by Page [8], there are many free Java simulation libraries available on the web, such as Simkit

[9], JavaSim [10], JSIM [11] and Simjava [8]. These free simulation libraries have therefore been used in the development of many simulation programs which have then been deployed on the web as Java Applets

Figure 2.2 shows a basic client simulation and visualization configuration where the simulation and visualization engines are loaded into a browser in the form of a Java Applet or a similar form such as Adobe Flash.

Figure 2.2. A Basic Client-side Simulation and Visualization Configuration

The advantage of using this approach is that there is no network latency [3] as no data is exchanged over the network during simulation. However, a significant disadvantage is that loading a Java Applet page usually takes much longer than a normal web page. There are also many other issues to address and resolve such as different versions of JRE (Java Runtime Environment) and memory leakage. Because with this approach all the work takes places within a browser, it might only be suitable for small simulation applications.

2.3.2 Server-side Simulation and Visualization

Figure 2.3 shows a typical server-side simulation and visualization configuration where simulation and visualization engines are installed on a server. Simulation requests can be sent from a web client via any form of web element. For example, a simulation request is submitted via an HTML form to a web server (e.g. an apache web server) which then passes the request to a simulation engine via an interface (i.e., CGI). After the simulation has taken place, a visualization engine on the server produces images or videos containing the simulation result that can then be embedded into HTML pages. Therefore, the result can be displayed on the user's browser. The major advantage of this approach is its ability to handle large, complex simulation [3]; the principal drawback is the network latency.

Figure 2.3. A Server-side Simulation and Visualization Configuration

2.3.3 Hybrid Simulation and Visualization

Figure 2.4 shows a typical hybrid simulation configuration in which some of the workload has been transferred to the client-side where the visualization engine resides. This approach combines the advantages of the first two approaches [3].

Figure 2.4. A Hybrid Simulation and Visualization Configuration

2.4 Web-based Simulation Technologies Overview

Fishwick [5] states that a web-browser has now become a platform for running applications, and cloud-based applications development is now the future for software development. This has come about as a result of the evolution of web technology and the number and variety of suitable languages and tools that have emerged with which to develop web applications. In this section, web-based simulation technologies, standards, and tools are reviewed and discussed.

2.4.1 Middleware of Web-based Simulation

Middleware is a term used in distributed computing. Middleware technologies help to increase the flexibility, extensibility, maintainability and reusability of distributed applications [3]. Web applications and Rich Internet Applications (RIAs) can both be considered to be distributed applications. Many researchers [12, 13] have created classifications for middleware technologies. Hurwitz [12], for instance, classifies the middleware market into six categories:

- Asynchronous remote procedure call (RPC)

- Publish/subscribe
- Message-oriented middleware
- Object request brokers (ORBs)
- SQL-oriented data access middleware
- Synchronous RPC

The first three are asynchronous protocols; the rest are synchronous. According to Kodali's [14] definition of asynchronous communication "a sender is not blocking until a reply has been given, when it calls a service or sends a message." Figure 2.5 illustrates the difference between these two types of communication.

Figure 2.5. Comparing Synchronous and Asynchronous Communication

Byrne et al. [3] summarize the common middleware used in web applications. A simplified version of this is shown in Figure 2.6. These technologies handle communication between a server and a client and ensure that the communication is platform independent. These standards and technologies are mainly defined or developed by organizations such as the

Object Management Group (OMG), the World Wide Web Consortium (W3C), Sun Microsystems, and Microsoft.

OMG	CORBA, CORBA Component Model,
SUN JAVA	RMI, EJB, J2EE, Sockets, JMS
W3C	Web Services, SOAP, RPC, XML
Microsoft	.NET, COM, DCOM, .NET Remoting

Figure 2.6. A classification of Middleware Technologies

OMG is a consortium that sets standards for distributed object-oriented systems. A well-known example of a standard set by OMG is Common Object Request Broker Architecture (CORBA) and its extension, CORBA Component Model (CCM). CORBA is one of the most popular commercial middleware standards [3] and enables communication between programs that have been developed using different programming languages on different platforms. Commercial and open-source implementations of CORBA are available worldwide. Examples of free CORBA implementations include Inprise Visibroker, Iona's Orbix, Orbacus [15] and OmniORB. Rodriguez et al. [16] give an example of using CORBA, Java and XML for a web-based multi-body simulation. Other examples where CORBA has been used for web-based simulation can be found in [10, 17-20].

Remote Method Invocation (RMI) is the basic distribution communication protocol from Sun Microsystems. The RMI communication protocol is used for applications developed using Java. Like CORBA, it supports multi-platforms because it runs on Java Virtual Machine (JVM) making it platform independent. Unlike CORBA, it does not support multiple programming languages. A Java technology called Enterprise JavaBean (EJB) can be used to interact with programs developed in other languages. EJB is one of the APIs of Java 2 Enterprise Edition (J2EE). EJB consists of RMI and CORBA interfaces [3] and can therefore be considered to be an extension to RMI, like CCM to CORBA. Other communication protocols supported by Java include Socket and Java Message Service (JMS). Page [8] gives an example of using RMI for web-based simulation with the SimJava simulation package.

Like RMI to Java, Component Object Model (COM) is Microsoft's basic communication protocol. DCOM, COM+ and .NET Remoting are additions to and variations of the COM protocol. Just as Sun Java has a J2EE framework, so Microsoft has a .NET enterprise framework which contains a list of communication protocols and APIs for developing enterprise applications. Harrell and Hicks[17] provide an example of using both CORBA and DCOM for implementing embedded simulation in enterprise applications.

J2EE platform and .NET framework are two of the most popular enterprise-level software development environments in the world. One of the key elements of the J2EE platform is Java Runtime Environment (JRE) which consists of Java Virtual Machine (JVM) and a

number of Java APIs. Java programs are first developed using Java APIs before being compiled and run on JVM. JVM makes Java programs portable so that they can be developed once but run anywhere. JRE is available for different platforms (e.g., Windows, Linux and Macintosh) and is also available in plugin form for different browsers. With JRE browser plugins, it is possible to run Java applications as Java Applets within a browser. According to James et al. [3], J2EE defines a set of specifications that can be implemented by other vendors such as IBM Websphere, BEA WebLogic, and JBoss, and these implementations are used widely to develop enterprise-level applications.

.NET framework has a runtime execution environment called .NET runtime which is only available on the Windows platform. However, it enables interoperability of programs that have been developed using different Microsoft programming languages including Visual Basic.NET, C# and Visual J# .NET, as well as the scripting language Jscript .NET. The advantages of J2EE are platform independence, multi-vendor support and a large number of tools and resources. The advantages of .NET are its support for multiple languages and its ability to seamlessly integrate Microsoft web services; the disadvantage is its single vendor support (i.e., Microsoft). The disadvantage of J2EE is its single programming language support (i.e., Java language).

The World Wide Web Consortium (W3C) is an international organization that sets standards for the World Wide Web. HTML, CSS, DOM, CGI (Common Gateway Interface), XML, SOAP, XHTML and WSDL are the common standards used in today's

web application development. XML and SOAP have been widely used in combination in many .NET and J2EE applications. SOAP [21] is an XML-based message exchanging protocol that runs over HTTP. CGI is a web technology on the server side that was formerly used to handle website requests. It is script-based and can act like a gateway between a web server and a software program. For example, in early web-based simulation a CGI script was used to deliver simulation data, including source code and simulation configurations, from a web server to a simulation program. Once the simulation had finished, another CGI script was used to generate images of the simulation result that could then be embedded into web pages. Several researchers have used this approach [7, 22-24] for web-based simulation. For example, Guru et al. [22] describe CGI being used to develop a toolkit for SIMAN simulation models. Lorenz et al.[7] present an example of using CGI and Java Applet to develop a simulation and animation environment for the GPSS simulation language.

2.4.2 Front-end Visualization Technologies

A normal web page presents information that is described using Hypertext Mark-up Language (HTML). A browser has a built-in HTML parser which is used to parse and display the HTML document. Every action a user performs within a browser (such as refresh, back, forward, click on a link) calls a new web page from a web server. Web pages can be generated in several different ways. The usual way is to keep predefined web pages on the server's disk so that whenever a request is made, the web server reads the page from the disk and returns it to the client.

Technology has evolved to an extent that scripting languages are now used to create dynamic web pages on the server-side. As summarized by James Byrne [3], some of the most popular server-side scripting languages are PHP, Perl, Python, ASP/ASP.NET, Java Server Pages(JSP), Adobe ColdFusion, and Smalltalk. This method of generating web pages is more powerful and flexible and many websites have been built this way as a result. However, a full HTML web page still needs to be returned to the client every time there is a request and this means that a browser has to be continuously refreshed with a new web page. Farrell and Nezlek [25] state that the major drawback of this approach is the redundant data that is being passed over the network and the increase in average wait time for a web page. However, server-side scripting languages enable dynamic web pages to be generated and this continuous refresh serves as a reminder to users that it is not a web application they are using but a web page.

Browser-side Technologies

Today's browsers are equipped with plugins, add-ons and extensions like JavaScript Engine, Flash, Silverlight, Unity Web Player and Java Runtime Environment (JRE). With these extensions and plugins, web browsers can support multimedia content such as flash, animation, video, audio, Adobe PDF, and Java Applet. Some of these extensions have now become a standard configuration of modern browsers (e.g., the JavaScript engine). A web page can be described with HTML and JavaScript and can perform certain actions without refreshing the entire page. An example of this is where JavaScript is used to verify whether a user's input is a number or a letter. In this way, a web browser becomes a platform for running a program developed using JavaScript language. Although the browser will still

need to be refreshed to display the content received from the server, it is, at least, on the right track towards becoming a web application. The situation has further improved with the advent of asynchronous JavaScript and XML (AJAX).

AJAX is a collection of technologies consisting of JavaScript, XML and DOM (Document Object Model). HTML DOM defines a standard mechanism for accessing and manipulating HTML documents. For example, a web page can exchange information with a server in XML format and HTML DOM is then accessed by some JavaScript programs to display information on a web page dynamically. Hence, a web page can be updated without refreshing the entire page, making it behave more like an application. As previously mentioned, there are other technologies like Flash, Java Applet, and Silverlight that can be used to create web applications. However, the main strength of AJAX is that it utilizes technologies that are widely available in today's web browsers without requiring the installation of any additional plugins.

Java Applet

AJAX rather than Java Applet is now the most popular approach used for developing web applications on the internet today [25] especially in the field of simulation. Java Applet is a piece of code that can first be downloaded then executed within the browser. It can be used as a client tool to provide interaction and visualization in web-based modeling and simulation. A great deal of research into web-based simulation has been conducted using Java Applet since the first presentation on the subject at the Winter Simulation Conference

in 1996. Chad [26] provides an example of using Java3D in Java Applet for visualization in web-based simulation. Miller et al. [27] give an example of using technologies such as Java Applet and JavaBean for JSIM web-based simulation. JSIM is built with JavaBean technology and is a component-based simulation package for quantitative numeric systems. Perkins et al. [28] present different approaches, such as a downloadable Java jar file, Java Applet, Flash, and Java web start, in an interactive simulation system for learning and teaching physics. Lorenz et al. [7] report an approach combining Java Applet and VRML-based 3D animation for visualization of simulation results in web-based simulation.

HTML5

Today's standard web browsers such as Internet Explorer8, Firefox, and Chrome are equipped with JavaScript engine and have HTML5 support by default [1]. With HTML5, web applications have better interaction and look more appealing. Hypermedia documents provided by traditional web pages contain text, image, audio, and video [4]. HTML5 makes it is easy to present information by providing its own tags for audio and video.

2.4.3 Rich Internet Application

A Rich Internet Application (RIA) [29] is a web application that shares many characteristics of a desktop application including an interface, usability, responsiveness, and undo/redo operations. RIA is delivered inside a browser, probably with browser plugins or using JavaScript. As summarized by Taivalsaaris [1], the most promising web application technologies are AJAX, Adobe Integrated Runtime (AIR), Adobe Flash [30],

Google Web Toolkit (GWT), Google Gears, JavaFX, Microsoft Silver light, and Ruby on Rails. These technologies can all be used for developing RIAs. In this section, these cutting edge web technologies and their use for RIA modeling and simulation have been reviewed.

JavaScript is a dynamic programming language and differs from traditional programming languages like Java and C++ in that it has no proper IDE (integrated development environment) and no proper debugging tools. Consequently, it is difficult to develop performance RIA applications directly with JavaScript. Google Web Toolkit (GWT) provides a way to speed up the development process by enabling developers to build complex web applications in programming languages (i.e., Java) with which they are familiar. Complex and high performance web applications are developed using Java language while GWT handles the compiling from Java into JavaScript language. The final applications are all released in JavaScript. Since the applications are represented in JavaScript and every browser nowadays has a JavaScript engine, this means that RIA applications can be executed in a standard web browser without the need to install additional plugins.

Google Gears is an extension of the Google core JavaScript library. It provides local storage and a database facility to enable some online files to be used offline. As a result, more powerful and more robust web applications can be developed. Google Gears, however, is not a complete solution for developing web applications and has to be used in conjunction with other technologies such as AJAX and GWT.

JavaFX is a web application development technology from Sun Microsystems. It is built on top of the Java core library and a user interface library called Java Swing library. JavaFX requires the Java Runtime Machine (JRE) plugin to enable applications to be run in a browser. Microsoft Silverlight is an application framework for developing flash-style web applications and usually also needs a browser plugin to run them within a browser.

XML is extensively used in developing RIA applications in areas such as communications and graphic interfaces. XML is a well-known technology for storing information and has been used to support other technologies like SOAP and AJAX. A technology called JavaScript Object Notation (JSON) has recently taken up its place as a data representation format. JSON is usually treated as an advanced version of XML in data representation, especially when working with JavaScript. JSON is a text-based open standard for representing data structure that can be transmitted between servers and web applications. Cui and Ni [31] note that JSON works much better than XML in AJAX and Gonzalez and Cuadrado [32] develop a neutral XML-based language for describing multi-body systems.

2.4.4 Database Technologies

A database is usually needed to store data for a web application. The database used most commonly with web applications is the relational database. There are a number of relational database management systems (RDBMS) on the market: MySQL, Microsoft Office Access, Microsoft SQL Server, PostgreSQL, Oracle Database, and IBM DB2.

MySQL is the most popular open-source DBMS used in developing web applications. Miller et al. [27] give an example of using MySQL DBMS for a web application. Microsoft provides two tools in RDBMS: Microsoft Access and Microsoft SQL Server. The former is file-based and can be stored on one's hard disk; the latter is like MySQL but is normally used for enterprise applications. Byrne et al. [30] present an example of using Microsoft SQL in a web application.

A database connector/driver is usually needed to talk to database systems. In the J2EE environment a connector called JDBC can be used to connect to a MySQL database. JDBC is easy-to-use and useful for small web applications. Because of its simplicity, however, users need to write every piece of every SQL query needed for operations such as Select, Insert, Update, and Delete. This makes it too time consuming and inefficient for large web applications. Another connector called Java Persistence API (JPA) is used to counter this problem. JPA matches database tables into programming objects in Java. As such, database operations become object manipulations in a programming language, which is an easier and more efficient way for developers to do data manipulation. JPA [33] also takes care of security and data duplication issues. JPA is a protocol rather than an implementation. There are several vendors that implement JPA: DataNucleus, EclipseLink, JBoss Hibernate, ObjectDB, and OpenJPA. Miller et al. [27] give an example in their research of using JDBC to connect to MySQL.

2.5 Modeling and Simulation Language and Tools

Overview

The development of computer simulation has resulted in it playing a more and more pivotal role in a wide range of fields such as physical systems modeling, manufacturing applications, semiconductor manufacturing, construction engineering and project management, military applications, logistics and supply chain management, transportation and traffic, business process simulation, and health care. Nance [34] states that simulation, as a problem-solving technique, even appeared before digital computers. According to Nance [35], computer simulation began during World War II and has since been used in areas like research, system analysis, education, training, and entertainment. In the manufacturing sector, for instance, 3D visual modeling software helps in design, analysis, and flaw identification. In logistics, simulation software such as Arena helps to find the optimum solution for a queuing problem. In physics, simulation software can be used to model physical systems such as electrical, car suspension and rocket systems.

As can be seen, simulation is an extremely broad area; almost anything can be modeled and simulated to some degree. As Nance [35] notes, simulation can be looked at from many different perspectives. For example, purpose of using simulation (e.g., analysis, training, research, entertainment, education and gaming), types of simulation models (e.g., discrete-event, continuous, and combined discrete-continuous), and simulation languages or tools (e.g., GPSS, Modelica, Arena, and Solidworks). Based on the perspectives of types of

simulation models, there is a review of modeling and simulation languages or tools within which different types of simulations and simulation languages or tools are discussed.

In some perspectives, modeling and simulation are two different concepts. Modeling is more about representing a physical system in a computer model while simulation is more about analyzing a computer model, i.e., generating and studying the simulation result. In this section, modeling and simulation are discussed together because most of the languages involved are capable of both modeling and simulation. Hence, in this thesis the languages are simply addressed as simulation languages. There are some simulation tools such as GPSS and ExtendSim that come with their own simulation languages. Usually these languages share the same name as the simulation tools since they are exclusively used by them.

2.5.1 Use Programming Language for Simulation

Modeling is about describing a physical system in a computer language and can be done in two ways [36-38]. The first is to use a traditional programming language such as FORTRAN, C, Java, and Simula to describe the physical systems. The earlier simulation (both discrete-event and continuous-time) models all used this approach. Simulation models developed using this approach can leverage the advantages of existing compiler technology for programming languages. Most of the simulation libraries developed using this approach are specialized for particular domains, but multi-domain support is not possible [36].

Java is a widely used programming language for developing simulation libraries, for example, JSIM [39] (i.e., a simulation library on the process interaction paradigm), Netsim [40] (i.e., a Java simulation package designed for simulation on the Internet by Tamie), SimJava [41] (i.e., a Java library introduced by McNab and Howell), JavaSim, Silk, EJS and Moose [6].

2.5.2 Emerging of Simulation Language

The second way to describe a physical system is to use an appropriate simulation language. System modeling uses a degree of detail, for example, a less detailed modeling of a car shows only the covering body whereas a more detailed modeling shows the internal components of the car as well, such as the engine, suspension system, brakes and wheels. However, the increase in the number of requirements or details to be modeled reduces the capability of the programming language or simulation libraries (that have been developed using a programming language) to fulfill their task. It is therefore necessary to design a simulation language specifically for modeling, such as Modelica for multi-domain physical systems and GPSS for discrete event system simulation [36]. One of the advantages of a simulation language over a programming language is that it can implement specific modeling techniques such as Differential-Algebraic Equation (DAE), and physical laws (e.g., energy conservation, Newton's law, and Kirchoff's law). Another advantage is that most simulation languages can insert graphic notation allowing simulation models to be

built using a graphic approach. However, these simulation languages usually lack existing compilers. Hence the simulation of models using this approach involves two steps:

- First, a parser is used to interpret and translate models into a language that can be compiled using native compilers such as C/C++ and Java.
- The next step is to invoke a native compiler to compile the translated code into machine code and then carry out the simulation.

Examples of this include ACSL models (that can generate FORTRAN representation), Simulink models (that can generate C representation) and Modelica models (that can generate C representation).

Generic modeling languages such as UML (Unified Modeling Language) and SysML (Systems Modeling Language) are potential alternative languages for use in simulation modeling. However these are abstract and are typically used to present information to humans, rather than to computers. They therefore require extra effort to implement. As their original purpose was for something other than simulation, it makes simulation modeling more difficult. Nevertheless, some research is taking place in this area. For example, Lei et al. [36] present a work that makes UML models capable of simulation.

2.5.3 Simulation language for Multi-Domain Physical Systems

As discussed earlier, modeling and simulation has been used in many fields, however, this thesis is concerned specifically with the modeling and simulation of physical systems.

Simulation models of physical systems are today more complex than ever before. For instance, a model can combine continuous-time and discrete-event behavior, or a higher level of discrete-event simulation with a lower level of continuous simulation [34], and a model can span multi-domains such as electrical, mechanical, hydraulic and control. Simulation languages such as CSSL and SLAM can model systems that combine both discrete-event and continuous-time behaviors. With few exceptions, these simulation languages are specialized in one distinct domain [42] and not able to model systems that integrate components from diverse domains such as electrical, mechanical, hydraulic and control, as commented by Peter Fritzson [43]. SLAM, for example, is used for network system simulation [36], VHDL-AMS and VerilogAMS are used for electrical systems, and SPICE is used for analog circuit simulation. The demand for modeling multi-domain systems is increasing. For example, a DC motor consists of electrical parts and mechanical parts as exemplified by Steve Miller [44]. Languages for modeling this type of system require a special design to ensure they have the capability of modeling complex physical systems. Currently, due to the increasing demand for complex physical systems simulation, new simulation languages and tools are starting to emerge.

General Requirements

Nance [34], Elmqvist [45], and Jeandel et al. [46] have all described the requirements for these kinds of simulation language. The most frequently argued and cited requirements are summarized below:

- **Combine Simulation Support:** a language must be able to describe complex systems involving both continuous and discrete-event behavior.
- **Multi-domain Modeling:** a language must be able to support the modeling of complex systems that span multi-domains.
- **Independent:** a language should have its own specifications and be independent of any existing simulation environments.
- **User-friendly:** description of the language should be meaningful and readable by human users; it should follow the general conventions of programming languages such as C, FORTRAN; and it should have documentation to track the versions using information, graphic information etc.
- **Computer-friendly:** a language must be able to be parsed and compiled on computers, that is, the language should define grammatical rules, implement semantics, and follow the common constraints of programming languages such as variable lengths, zeros, and infinity.
- **Reuse:** a language must be highly flexible with respect to the model hierarchy. Assembling and disassembling a complex model should be as easy as playing with Lego blocks.
- **Repeatable:** rerun models, or repetition of the validation process should be supported without having to repeat the modeling process.

Physical Rules

The basic concept behind languages that support multi-domain modeling is that they describe a physical system using basic physical rules. No matter which domain this physical system is from, it must obey the basic physical rules such as:

- **Kirchhoff's law:** at any point of an electrical system, the sum of currents flowing into one node is equal to the sum of currents flowing out of that node
- **Energy conservation rules:** energy flowing in and out must be equal.
- **Newton's laws:** states and forces applied in one direction at a node must be equal to zero.
- **Fluid:** the amount of fluid flowing into a node must be equal to the amount of fluid flowing out of that node.

Broenink [47] summarizes the modeling languages that have been developed based on these rules as ASCEND, Dymola, gPROMS, NMF, ObjectMath, Omola, SIDOPS+, Smile, ULM and VHDL-AMS. However, most of these are environment-dependent, which means that models built with one tool cannot be used by other tools. A language called Modelica [45] has been developed in an international effort to facilitate the sharing and exchange of models and model libraries between tools. It has been developed with the aim of creating a unified modeling language for complex physical systems spanning multi-domains.

2.5.4 Modelica Language

Modelica [43, 48-53] is an equation-based, a-causal, object-oriented modeling language with ordinary differential and algebraic equations, using object-oriented constructs to establish a uniform representation of physical system models from multiple domains. Examples include systems containing mechanical, hydraulic, thermal, control, electrical, electronic, electric power or process-oriented subcomponents.

The Modelica project was initiated in September 1996, based on the PhD thesis on Hilding Elmqvist [45]. The development of Modelica was based on the experience accumulated with existing object-oriented, a-causal simulation languages: Allan, Dymola, NMF, ObjectMath, Omola, SIDPS+, and Smile. Modelica is an open-source textual-based language rather than a graphic tool and this definition has several advantages:

1. It makes the Modelica community focus on defining semantics and specifications.
2. It encourages various professional software vendors to develop compilers and graphic tools.
3. It encourages contributions from a large body of users (i.e., collective intelligence).

Modelica specifications are managed by a non-profit organization called the Modelica Association [54]. The Association not only develops and refines Modelica specifications but also develops the Modelica Standard Library (MSL). MSL includes packages from multiple domains such as mechanical, hydraulic, thermal, control, electrical, electronic, electric power, Math, Fluid, and Media. MSL keeps growing, and the current version, at the time of writing, is 3.2. The current version of Modelica specification is 3.3 and was released on June 06, 2012.

Modelica is Open Standard; hence other free and commercial Modelica libraries are available to use with it, like the Motorcycle Dynamics Library in Modelica [55] which has been developed to enable the dynamic simulation of a motorcycle within a Dymola environment. One example of using the Modelica network library to simulate and evaluate a networked control system is presented by Liu and Frey [56]. Martin-Villalba et al. [57] present a third party Modelica library (i.e., VirtualLabBuilder) that allows virtual interactions with the simulation model: virtual feedback is provided through graphics and parameters can be easily adjusted.

Modelica's object-oriented characteristics make it highly flexible and able to support encapsulation, inheritance, and hierarchy [58]. Modelica uses differential algebraic equations to describe the continuous-time behavior. Equations in Modelica are represented a-causally, i.e., an equal sign "=" in the Modelica language means "real equal" in mathematics. This is different from the meaning of "assignment" in computing languages (C, Java) and makes it possible to dramatically reduce modeling complexity when programming, since one Modelica equal statement can replace three assignment statements. Furthermore, a-causal modeling simplifies the incorporation of physical rules. Physical rules such as Newton's laws of motion and Kirchhoff's law can be described using a single statement. As discussed in section 2.5.1.3, physical systems from different domains all obey the basic physical rules and laws and thus make it possible to model multi-domain physical systems. Modelica has the following basic elements [58]:

- The number of equations must be equal to the number of variables to enforce a valid simulation solution.
- The sum of the input on one node must be equal to the sum of the output.
- It follows the same constraints and conventions of normal programming languages, for example, it limits variable name lengths and has divide by zero violation, and infinite loop violation.

These basic elements demonstrate that Modelica is a suitable modeling language for physical systems [52].

Nevertheless, Modelica is one of the most popular modeling languages used in many different fields of industry such as fluid systems, automotive applications, mechanical systems, thermo-fluid, and energy systems. In the automotive sector, many of the leading automotive OEMs including Ford, General Motors, Toyota, BMW and Daimler are all using Modelica.

2.5.5 Other Languages and Tools

There are several languages that have been developed based on the ideas of a-causal, equation-based and object-orientation, such as Simscape [44] and Acumen [59]. These and other simulation languages (SIMAN and XML) popularly used by researchers are discussed below.

- **Simscape** is a MathWorks modeling language or tool. It is very similar to Modelica in that it is object-oriented, a-causal, and equation based, and it supports multi-domain modeling. Because it is a MathWorks product, it can be used with MATLAB and supports C-code generation.
- **Acumen** is a modeling language inspired by Modelica and used to model hybrid systems of highly dynamic cyber physical systems.
- **SIMAN** [22] is a simulation language that allows users to simulate discrete and continuous systems. Guru presents a web-based interface with which to edit, simulate and store text-based simulation models in SIMAN.
- **XML** has been used extensively to represent data on the web. Fishwick [60] gives an example of using XML to describe simulation models. Gonzalez and Cuadrado [32] present a neutral and extensible XML-based language called MbsML for describing multi-body system models and related information. Adrian Pop et al. [61] present an XML representation of Modelica called ModelicaXML used in the development of OpenModelica projects.

The literature on physical system modeling tools, for example, Simulink, Scicos, and FlexSim, that enable users with fewer programming skills to do modeling has also been reviewed.

- **Simulink** is a well-known tool from MathWorks for physical systems modeling. It requires a deep understanding of the physical system including internal flow

control and energy transfer. There is no web-based version of Simulink currently available on the Internet.

- **Scicos** is another tool that supports physical systems modeling. Its name has recently been changed to Xcos. By the time when this thesis was written, there is no web-based version of scicos currently available.
- **FlexSim** is a modeling tool for manufacturing, logistics, and transportation. The current version of Flexsim is still a windows-based simulation software.

There are tools available that use computer aided technologies and that have fancy graphics and visualization to help people in computer-aided design, computer-aided engineering, and computer-aided manufacturing and product data management. Examples of these include Adams, Cosmos, WorkingModel and Visual Nastran. These tools usually define their own language to describe the model and some of them also show some degree of simulation capability:

- **Adams** is for modeling multi-body physical systems. It is the most widely used multi-body dynamics and motion analysis software.
- **Cosmos** is an extension of Solidworks, and is suitable for stress/strain analysis.
- **WorkingModel** is an engineering simulation software product. Virtual mechanical components like springs, ropes, and motors are combined with objects in a 2D working space. After running a simulation, the program simulates the interactions of a model's parts and generates graphic views of motion and force on any chosen element in the project.

- **Visual Nastran** is a graphic based 3D modeling and simulation software like Cosmos, however it is more domain-specific and is more powerful in simulation and animation.

2.5.6 Functional Mock-up Interface

As described above, there are many modeling and simulation tools, and each of them uses its own modeling language which makes the exchange of simulation models becomes extremely difficult. Researchers from MODELISAR project [62] define a Functional Mock-up Interface(FMI) standard for model exchange by using a combination of XML files and C-code. This interface standard [63] supports exchange of models that are described by differential, algebraic and discrete equations with time, state and step-events.

A FMI model is distributed in a zip-file that contains a XML file, a C-code file, an icon image file and etc. The XML file contains the definition of all variables in the model and other model information. All model equations are provided in the form of c-functions. The FMI defines an open interface that can be implemented by other simulation tools in order to import or export FMI models. A FMI model is independent of any target simulation tools because it does not use a tool specific language or structure as in other approaches.

FMI has been supported on many simulation environments such as AMESim, CATIA, ControlBuild, Dymola, JModelica.org, OpenModelica, Mworks, Matlab, MapleSim, SimulationX, Simulink and etc. A full and up-to-date list can be viewed at FMI website

[64]. Thiele and Henriksson [65] present an example of using FMI in AUTOSTAR simulation environment. FMI standard for model exchange is utilized in the context of AUTOSTAR software development. Automatic transformations feature between the XML schemas of FMI and AUTOSTAR standards are implemented.

2.6 Web-based Simulation Tools and Other Design Considerations

In this section, an overview of existing web-based simulation tools for physical system is presented. Secondly, based on the features of web-based simulation, specific topics such as block-based modeling, model repository, compiler design, and hardware-in-the-loop (HIL) modeling and simulation are reviewed and discussed.

There are some other web-based simulation tool as well. CyDesign [66] is a model-based design optimization platform with integrated requirements management that enables organizations to design complex systems with better and faster results. The initial design objective of CyDesign was to provide a web-based environment for gathering design requirements. After the acquisition of Delta Limited [67] in 2012 which is the company who develop the commercial Modelica simulation tool-Vertex, another objective of CyDesign is to integrate the existing platform with a commercial, web-based Modelica simulation environment. CyDesign web application utilizes web utility player [68], which

requires additional browser extension to be installed. However, this commercial platform is announced to be released in 2013. Shi et. al. [69] present an internet-based electrical engineering virtual lab (iEEVL) using Modelica for unified modeling. It uses XML to represent and exchange information and is only capable of modeling and simulating for electrical engineering domain. iEEVL has a graphic interface on which Modelica components appear as graphic blocks. There are some other web-based simulation platforms which use a different approach: text-based representation for Modelica models. For example, UNVirtualLab [70], OMWeb [71] and DrModelica [72]. These platforms are all based on OpenModelica compiler. Another web-based simulation environment called WebMWorks[73] also caught researchers' attention. WebMWorks is a web-based modeling and simulation environment based on MWorks simulation engine. It uses XML for exchanging data between web-browsers and the server. It is said to support online model sharing. A similar online sharing and computing platform is NcLab [74]. NcLab is a commercial cloud-based web computing platform to provide web computing services for Matlab, Mathematica, MathCAD, and Finite Element (FEM) software.

2.6.1 Object-Oriented Vs. Process-Oriented Modeling

Lego is a popular construction toy that is flexible, portable, and reusable. With Lego small bricks (objects) can be used to create hundreds and thousands of new objects. There is also a robotics edition of Lego called Mindstorms with which one can make motorized moving objects, like robots, vehicles and toy dogs. In the software world, the object-oriented approach uses the same concept as "Lego". However, object-oriented programming is restricted to use by programmers because it is a specialized area of work, requiring

specialized skills. A research group from MIT has solved this problem for ordinary users by introducing a graphic programming tool called Scratch. Scratch, as shown in Figure 2.7, leverages the beauty of the object-oriented programming language but enables individuals (e.g., kids) to create animations using basic operations such as drag-and-drop and rearrange programming blocks. Each programming block has its own connectors like the connectors on Lego bricks and only matching blocks can fit together.

Figure 2.7. “Lego-like” Programming for Kids

In the world of simulation, people have been looking for tools that allow complex simulation models to be assembled dynamically using small components, and there are two existing approaches: object-oriented and process-oriented.

The terms object-oriented and process-oriented are used in simulation tools. Simulation tools often have block diagram representation, which enables engineers to build simulation models without writing scripts [37]. For example, the Arena simulation program has a graphic library containing blocks of entity, queue and activity. Users can drag-and-drop these blocks and connect them using graphic wire on a canvas to create a visual model. Each small block looks like a small object. However, a simulation tool with block representation does not mean it is using object-oriented modeling. For example, some tools such as Arena and Simulink have block representation but they use process-oriented modeling. Another misunderstanding is that simulation tools developed using an object-oriented programming language use object-oriented modeling [38]. For example, SIMAN and SLAM are described using an object-oriented programming language (i.e., C++) but they use process-oriented modeling.

Object-oriented modeling is sometimes referred to as component-based modeling. Component-based modeling is more intuitive because it is designed to match the way people perceive the physical world: everything is a component and a complex component is composed of small components. A complex component can be easily broken down into reusable small components. Process-oriented modeling is often signal-based or input-output based. Early simulation tools such as Simulink and Arena were process-oriented [44]. As shown in Figure 2.8, an example from the Wolfram SystemModeler illustrates the differences between these two concepts. LHS (i.e., left hand side) is an electrical system that is modeled using the object-oriented approach; RHS (i.e., right hand side) is the same system modeled using the process-oriented approach. Object-oriented modeling is also

known as component-based modeling, and process-based modeling is also known as block-based modeling.

Figure 2.8 Object-Oriented Modeling vs. Process-Oriented Modeling (This picture is from the Wolfram SystemModeler Website)

Advantages of Object-Oriented Modeling

Many researchers [38, 75, 76] have discussed the advantages of object-oriented modeling. Roberts and Dessouky [38] summarize them as: intuitive, offering reusability of software and design, faster development, increased quality, easier maintenance, enhanced

modifiability, and reduced development risks for complex systems. Objects are how humans perceive the physical world. The world consists of objects, for example, a mug or a spoon. Hence modeling physical systems (e.g., a car) into objects is more intuitive. Reusability is one of the key features of object-oriented modeling and extensive research has been done in this area. For example, Jeandel and Boundaud [46] state that, reusability requires a model to be represented in as general a way as possible. Lei et al. [36] develop a meta-model based representation to achieve the reusability of simulation models. Because of these advantages, there has been an enormous increase in research and development into object-oriented simulation tools [38] since the mid-1980s. As a result, many object-oriented simulation languages have been developed. Roberts notes that the first object-oriented simulation language was Simula [38]. Smalltalk is another one that is based on Simula. Modelica [48] is an object-oriented simulation language designed for exchanging simulation models.

Object-oriented modeling has many advantages and has become the standard for physical systems modeling. However, there are some areas, such as control systems [44] that still call for process-oriented modeling.

2.6.2 Model Repository for Storing and Exchanging models

As a result of the increase in the use of modeling and simulation, it has become important to ensure that modeling and simulation languages or tools meet certain quality and

efficiency criteria. Moreover, Jeandel et al. [46] note that modeling and simulation would be more effective if models could be stored and exchanged without losing information.

With the development of the Internet, the web has become a medium for publishing information as well as a place for sharing and exchanging objects. James Byrne et al. [3] state that a model repository is a server-based centralized repository that can be used to store simulation models. James Byrnes states that:

“This allows for the ability to rapidly disseminate models, results and publications. Just as a web document may be hyperlinked to several other Web documents, models may be linked together to form a model federation.”

In the panel discussion at the 1998 Winter Simulation Conference, Fishwick [77] argued that a critical problem in the field of computer simulation is the lack of published models and objects to facilitate model reuse. One of the benefits of publishing models on the web is that other authors can download and assemble them to make a composite model.

Fishwick [78] discusses various issues around publishing digital objects and models on the web for reusing. One of the most discussed problems [77] concerns the quality control of published components from the web. Nance strongly supports the standard mechanism for controlling the quality of publishing models or objects on the web. Paul, on the other hand, had a different view. Paul thought that ensuring the correctness of published components would take too much time and money, and sometimes the answer might even be

indeterminately wrong. In his view, massive and rapid search, adopt, and experimentation over a large number of components would dramatically save time and trouble. In the end, Fishwick comments, quality control must be maintained where it is necessary, depending on the overall objective of the simulation. For example, component quality in simulation for entertainment is a less sensitive issue than in simulation for analysis and rocket science. However, the war over the quality control of published components is not over and has been extended to other areas, for example, the application stores of the Smartphone. Apple controls the applications that can be run on IOS while Google allows any applications to run on Android. There is no way of telling which way is best because both approaches are reasonably successful.

Fortunately, web-based modeling and simulation [77] has a bright future. It is agreed that simulation components and models need to be published and exchanged, and then reused to increase productivity. Reusability is best achieved via a model repository. A model repository is a central place where models are collected, organized, and re-used. Several different approaches to developing a web-based model repository are suggested by the research. Iazeolla and Ambrogio [79] built a model repository for reusing simulation models expressed in QNAP2, a generic language for discrete-event simulation using technologies such as Java, CORBA, and HTML. A text-based model repository [22] for simulation models in the SIMAN language is presented using CGI and HTML by Guru. A kinetic model database [80] is presented using Java and Mathematica technologies. Perkins et al. [28] present an online interactive simulation system called Phet. This has a website

that serves as a repository of physical simulation experiments that have been created and tested by the Phet research group.

2.6.3 Compiler Design of Modelica

A physical model described using the Modelica language is called a Modelica model. In order to simulate a model written in the Modelica language, a compiler is needed which can transform a Modelica model into an executable file. As described in section 2.5.1.3, Modelica is a modeling language with its own syntax, grammar and semantics. The simulation process usually takes two steps:

- Firstly, parse and translate a Modelica model into a language that can be compiled using existing native compilers such as C/C++, Java, Python compilers.
- Secondly, invoke native compilers to compile the translated code into machine code to simulate.

The usual approach taken with existing open-source Modelica Compilers [81, 82] is to parse and convert Modelica code into C code, which is then compiled into binary machine-code with a C compiler, such as GCC on Linux or Visual C++ on Microsoft Windows.

With native compilers, the second step of making a Modelica compiler is almost an automated task; hence discussions and reviews in this section focus on the first step. A Modelica compiler consists of three stages as depicted in the dashed boxes in Figure 2.9. The first stage is to use Modelica source code as the input, perform lexical and semantic

analysis, and generate a flat Modelica model. The flat Modelica model is then passed to an equation analyzer and optimizer to sort and eliminate redundant equations. The Code Generator takes the output from the equation optimizer module as input and generates the corresponding C code.

Figure 2.9. Typical stages of Translating and Compiling a Modelica Model

There are several open-source compiler development tools available on the market. For example, JastAdd [83] (which is used by the JModelica.org compiler) and ANTLR [82] (which is used by the OpenModelica compiler). There are some other tools that are often used to develop compilers, such as JavaCC [74], Beaver [84], SableCC, and JavaCUP.

Building an Abstract Syntax Tree (AST)

and Beaver) often include modules that can help to develop a lexical analysis parser. However, JastAdd is an exception in that it uses other compiler generators such as Beaver, JavaCC, and JavaCUP to generate AST. JastAdd is an extension of JavaCC and can greatly enhance the manipulation power of an AST. Information can be freely added to the AST and obtained from it [84], which is particularly useful for the following steps such as semantics analysis and inheritance flattening. An in-depth understanding of the grammar of the specified language (as shown in section 6.3.2) is also required in the process of building the AST tree.

The constructed AST is then used for semantic analysis and the inheritance flattening processes. Semantics analysis requires knowledge of the full Modelica specifications. Broman [86] notes that Modelica semantics are defined in a natural language, and because of that individual users can interpret it ambiguously. Additional techniques and tools have been developed to tackle this problem:

- **Modelica XML**: implementation of a compiler usually requires data manipulation and interaction among different tools in the framework. Adrian Pop et al. [61] propose an XML representation for Modelica models.
- **ModelicaDB**: like any other object-oriented language, the Modelica language supports inheritance and composition. For example, the Modelica Standard Library (MSL) involves a lot of model reuse, inheritances and modifications. For efficient searching in a large base of simulation models, a searchable Modelica database is required. For example, the ModelicaDB [87] proposed in the OpenModelica project.

- **Modeling Kernel Language:** a new language called the Modeling Kernel Language (MKL) [86] has been introduced to allow formal reasoning and increased accuracy in designing the semantics of the Modelica language.

Existing Modelica Simulation tools

There are several commercial Modelica simulation tools available, such as Dymola [88], ModelicaSDK [89], MOSILAB [90], AMESim [91], MapleSim [92], MWorks [93], MathModelica [94], and SimulationX [95]. There are also some open-source compilers like JModelica.org [81], Modelicac [96], Scicos [97] and OpenModelica [98]. JModelica.org platform is an open-source compiler and optimizer for Modelica and Optimica, which is an extension of Modelica dedicated to dynamic optimization problems. Modelicac is a Modelica compiler that implements only a subset of Modelica semantics. For example, Modelica control semantics by means of discrete variables, algorithms, and blocks is not handled by Modelicac. Scicos [97] is another free Modelica simulation environment that has been used by researchers [99] to handle hybrid model simulation. Scicos is a component-oriented editor and simulator of Scilab. Their relationship is similar to the relationship between Simulink and MATLAB. As mentioned, Scilab includes the Modelicac compiler that only supports a subset of the Modelica language. The original Scicos [100] is not well suited to physical component-level modeling as it uses a process-oriented approach. An extension to Scicos that makes physical component-level modeling possible is developed using Modelica.

MWorks [93] is a Modelica IDE for the modeling and simulation of multi-domain physical systems and is developed by Huazhong University of Science and Technology in a joint effort with Suzhou TopRing Soft.& Contr. Co. Zhao et al. [101] give an example of using MWorks for the simulation of aircraft thrust reverser hydraulic systems.

There is some research that compares the quality of existing Modelica compilers. Sinha et al. [102] and Frenkel et al. [103] have both explored this area and made valuable contributions. The related statements have been reviewed and summarized:

- **Language Features:** to test the capability to handle different statements such as extend, import, declare, annotation, and array.
- **Symbolic Manipulation Power:** to test which simplifications and manipulations are undertaken by the compiler in order to improve simulation speed
- **Numeric Solver Robustness:** to test with complex models that might feature high indices, inconsistent initial values or singularities
- **Compiling and Simulation code Performance:** to test a set of predefined models and measure the time taken for translation or simulation respectively

2.6.4 Hardware-In-Loop (HIL) Modeling and Simulation

Hardware-In-Loop (HIL) simulation is where some components of the software simulation model are replaced by the actual hardware. It is a useful tool for testing and validating hardware components and is also used in control systems. The major difference between a HIL simulation operation and a normal computer operation is that a HIL simulation operation focuses on deadlines rather than general throughput. Normal PC operating

systems are concerned with throughput and as long as the overall throughput meets the performance requirements then it is acceptable to users. In HIL simulation, every single event must respond immediately in order to meet the deadline. In this case, a real-time operation is needed. Real-time operations can be guaranteed either by a real-time hardware device or a software component. If the real-time operation is guaranteed by a hardware device, it is called hard real-time (a.k.a., hard HIL). On the other hand, if the real-time operation is guaranteed by software such as a real-time operating system (RTOS), it is called soft real-time (a.k.a., soft-HIL). Real-time operating systems guarantee that deadlines can be met deterministically or statistically. Key factors in an RTOS are, therefore, low interrupted latency and high timer frequency which guarantee minimal thread switching latency. However, if the time requirement is not so critical, a software component can even be used to synchronize the hardware clock time on the external hardware with the digital simulation time on the computer. Ebner et al. [104] present an inter-process communication method called Named Pipe to address the synchronization issues in HIL operations.

RTOS

Real-time machines are usually expensive and PC-based soft real-time systems have become an alternative [105] for normal soft real-time operations. Hence, a lot of research has been done with real-time operating systems such as QNX, RT-lab, RTLinux, VxWorks, xPC Target and Windows CE. These are hard real-time operating systems available commercially and they usually operate on their specific devices. A normal open-source

Linux system can easily be patched to support real-time operations and this is the most cost-saving method for soft real-time HIL simulations. In fact, Linux has been chosen as the simulation platform for most research because of its flexibility, stability and cost-efficiency [106]. Abourida et al. [107] provide an example of using RT-Lab for a real-time HIL simulation.

HIL Simulation with Modelica

For real-time HIL simulation, Simulink is one of the most popular tools used for HIL control simulation. Muller and Waller [108], Gianni Ferretti et al. [109] provide an example of HIL simulation using Simulink. For real-time HIL simulation using Modelica, most users still rely on commercial Modelica simulation environments such as Dymola. Otter et al. [110], Elmqvist et al. [111] and Winkler and Guhmann [112] give examples of using Dymola for HIL simulation.

Dymola is an expensive commercial software program and not flexible enough to be easily modified for experimentation and research. As a result, some researchers such as Jugo [99] Bucher et al. [113] and Najafi et al. [97] have turned their attention towards other free Modelica simulation environments, for example, Scicos. As mentioned, Scilab, of which Scicos is a component-oriented editor and simulator, includes a Modelica compiler that only supports a subset of the Modelica language.

Modelica Hardware Drivers

The implementation of a soft HIL simulation involves three parts: a hardware driver, an interface to a simulation environment and a communication channel between the simulation tool and the hardware driver. Previous developments concerning devices driver support in Modelica include Bellmann's ExternalDevices library [114] and Elmqvist et. al's Modelica_EmbeddedSystems library [115]. Modelica Association unified, improved and extended from these libraries and released a free library called Modelica_DeviceDrivers [116] for interfacing hardware drivers to Modelica models. This library is cross-platform and supports soft HIL simulation. The current version of this library has support for Keyboard, Joystick/Gamepad, 3D connexion Spacemouse, shared memory, AD/DA converters and other devices. This library also supports the well-known Linux control and measurement device interface library-Comedi library [117]. High-level interfaces of these hardware devices on simulation tools are provided via Modelica blocks, which means users can interact with these devices via graphic Modelica blocks. With this library, the power of Modelica in the area of HIL simulation is improved significantly.

2.7 Web Application Deployment

2.7.1 Limited Computing Resources Issues

Hosting a web application consumes hardware, software and manpower resources. Hardware resources including CPU, memory, bandwidth, storage and I/O capacity are

usually limited. These limited resources are shared by a number of users at the same time, and this number can vary from a single digit to tens of thousands. For example, a personal website might only have a few visits every day, while a university website may have thousands of visits every hour. No matter how many visits a website (or a web application) receives, resources are always limited and should not be wasted.

CPU and Memory

CPU [118] is the short name for the central processing unit which carries out a program's instructions to perform the basic arithmetic, logic and input/output tasks of the system. Every program such as word processor, instant messenger and web server requires the CPU to carry out instructions to perform tasks. Hence, the CPU is the most critical resource. CPU is described in terms of clock-cycle; the higher the clock-cycle, the shorter the processing time. Memory is known as Random Access Memory (RAM), which is storage the CPU uses to perform tasks. The more RAM a computer has the better its performance will be.

Bandwidth

Bandwidth [119] is a term used to describe the data transmission rate of a communication link. It is measured in bytes per second, for example, 1000MB/s. In practice, bandwidth has two parameters: peak bandwidth and bandwidth quota. Peak bandwidth is the maximum speed a network link can carry data, and is measured per second. Bandwidth

quota is the maximum amount of data allowed per month. On a network, the term bandwidth is used in two places: the link between a server and the internet, and the link between a client and the internet. When deploying a web application, it is not possible to control the bandwidth on the client-side but it is possible to choose the bandwidth on the server-side. Hence the design and development of a web application considers the average bandwidth of the target clients. Moreover, it is important to take into account the server-side bandwidth when choosing a hosting plan with a web hosting provider.

Storage and I/O capacity

Nowadays, the price of hardware storage (e.g., hard-drives) has dropped dramatically. For most web applications storage does not seem to present a problem. However, if a web application consumes a lot of storage space (e.g., a video streaming application, a photo sharing website) then storage space becomes a critical resource. Another term related to storage is I/O capacity and refers specifically to the input/output capacity of hard drives. It is measured by the number of input/output operations per second. The traditional hard-drive disk [120] contains a rotating disk coated with magnetic material and a magnetic head used to read data from and write data to the surface of the rotating disk. Hence the data seeking speed is primarily related to the I/O capacity of that device. The data seeking speed is usually measured in rpm, or revolutions per minute, (e.g., a 7200rpm hard disk). A new type of storage device called Solid State Disk (SSD) can provide much higher I/O capacity. The mechanism of a SSD drive is quite similar to RAM. However, the price of SSD is still relatively high.

Virtualization

The virtualization of computing resources is the process of building a virtual machine of all available hardware resources which then appears as a single machine to end users. Virtualization is a term used frequently in distributed and cloud computing. In distributed computing, virtualization is a technology that combines different computing resources (e.g., several personal computers) making it look like a single computer to the end user. In cloud computing, virtualization is not only used in combining but also in splitting resources. For example, a powerful computer is comprised of multiple-core CPUs, large RAM, high bandwidth and large storage devices. Virtualization is used to divide it into multiple smaller instances for optimum utilization of resources.

A web application requires a certain amount of computing resources so that when deployed it can meet its performance and scalability requirements. However, performance and scalability are not the only issues: compatibility and security also need to be addressed.

2.7.2 Performance and Scalability Issues

Performance is an application's ability to execute to an acceptance level within a given time-span. Scalability is the ability of an application to maintain a certain level of performance when there is a substantial increase in usage [119]. These two terms are quite subjective. For the same web application, different users will have different acceptance

levels, and even the same user will have different acceptance levels for different web applications. For example, if you consider action response times of a non-profit organization website and a corporate website, users on the former are willing to wait much longer than users on the corporate website. A web application's performance is difficult to quantify because it depends on a variety of factors such as server infrastructure, application business logic, data logic, and presentation logic on the client- side.

Server infrastructure consists of software (such as webserver software, database software, and operating system) and hardware (CPU, memory, bandwidth, I/O capacity). Theoretically, in terms of performance and scalability, it is always the case that better hardware (i.e., more CPU cores, higher CPU clock frequency, more memory, and larger bandwidth) yields higher performance and better scalability. However, in reality, it is not always practical for reasons of cost-efficiency (i.e., the return on investment ratio). Determining what is the best computing resource solution for a new web application is a difficult task for several reasons. Firstly, utilization of the web application is difficult to anticipate at the outset, and secondly, utilization of the web application changes dynamically over time.

Therefore, the performance of a web application must be monitored in real time. Whenever performance degradation occurs, additional computing resources must be added and this requires a web application to have a flexible structure. For example, a web application that requires a lot of computing power to process users' data can be designed to separate the application computing server from the web server as shown in Figure 2.11. In this way,

additional application servers can easily be added in or removed from the system according to the traffic load.

Figure 2.11. A Typical 3-tier Web Application Server Architecture

There is one problem with this architecture. Monitoring a system's performance and adding or removing an application server accordingly is not that easy, especially when an application is hosted by a web hosting provider. Even with self-hosting, adding a new server to the existing system is still a time consuming task. It would probably take one week to purchase, configure, test and install the new server. However, cloud computing services such as the Amazon Web Service [121] and Google Compute Engine have built-in support (such as auto-scaling, elastic storage, cloudwatch and so on) for addressing these issues. With cloud computing, a new application server can be added into the existing

system within minutes, which can save web developers a tremendous amount of time and allow them to focus on designing the front-end and on critical thinking.

One can also adjust the business logic of a web application to improve performance. For example, the code can be properly tuned to reduce the number of disk reading/writing operations to save CPU and I/O time. Rubio [119] posits that tuning the business logic might produce only a negligible performance improvement as compared to adjusting the configuration of a web server.

However, the front-end logic must still be properly designed. With the emergence of HTML5 and RIA, a large number of tasks are performed on the client-side. For example, a web application with a graphic 2D canvas can allow users to draw different shapes within the browser. A web application can provide undo and redo capability on the browser and a web application can process data into drawings without contacting a server. However, the browser is still a thin client with limited resources. For example, there is a limit to the amount of physical memory on a client machine that a browser can use. It has been discovered that when a web application takes 400MB of a browser's memory, the responses to users' actions do not happen simultaneously (e.g., it takes more than 1 second). This makes tuning a web application's front-end logic of the utmost importance.

Databases can also be used and managed to address performance and scalability issues. For example, if an application requires a lot of reading operations on a database server, one can duplicate the database to use a master-slave structure as shown in Figure 2.12. Data can be updated from the master database but read from both master and slave databases. The slave database is synchronized with the master database. If the number of reading operations continues to grow, the number of slave databases can be increased accordingly.

Figure 2.12. A Master and Slave Database Structure

Other methods that can help tune an application's performance are caching and mirroring.

2.7.3 Browser Compatibility Issue

Another issue to consider when deploying web applications is browser compatibility. Different browsers and even different versions of browsers (i.e., Internet Explorer) have incompatibility issues like different JavaScript engines and different parsers for HTML and CSS. Farrell and Nezlek [25] state that it is a long and difficult process for a web application developer to handle these incompatibility issues manually. Luckily, a variety of frameworks have been developed to take care of the complexity of these incompatibility issues. Farrell and Nezlek [25] provide two examples of such frameworks, namely Prototype and Dojo.

There are some other web application deployment issues such as fault tolerance [122], security, and massive computing [123], however, due to the scope of this thesis, they have not been covered here.

2.8 Conclusion

This chapter presents an overview of relevant literatures starting from web-based simulation architectures, web-based technologies, simulation languages and tools. At the end, the deployment of a web application is presented and discussed. Specifically, Hybrid simulation and visualization architecture leverages the power of server-side processing and browser no latency processing. Many web-based simulation technologies such as, middleware, front-end and database technologies have been studied to prepare for the development work. Modeling and simulation section reviewed different modeling and simulation languages for physical systems and discussed specific topics such as block-

based modeling, model repository, compiler design, and hardware-in-the-loop (HIL) modeling and simulation. Modelica is still one of the most popular modeling languages used in many different fields of industry such as fluid systems, automotive applications, mechanical systems, thermo-fluid, and energy systems. Last section discusses web application deployment issues, especially, Amazon Web Services (AWS) has been discussed.

Chapter 3

A Knowledgebase Web Platform for Physical System Modeling and Simulation

3.1 Introduction

As discussed in Chapter 2, computer technologies and Internet innovation have had a significant impact on the simulation field. With a computer, users can safely perform simulation experiments that cannot be done in the laboratory either because of the cost or the inherent risks. The Internet makes a lot of information available to users from anywhere and at any time. In this chapter, an approach to experimentation that combines the advantages of both computer simulation and the Internet has been presented.

Chapter 3 discusses research issues in areas of a knowledgebase web platform of physical systems, simulation experiment learning, and information representation on the Internet. In the final part, it proposes a knowledgebase web platform for physical systems modeling and simulation (<http://www.visualphysics.net/pweb>).

3.2 A Knowledgebase Platform for Physical Systems

Today, the Internet is widely deployed and used. The extensive use of the Internet provides the opportunity to access vast quantities of information freely, but it also poses problems. For instance, a person searching the Internet for information and understanding of a two degree-of-freedom (DOF) system can easily become distracted by other things like links and pictures. Another problem is information overload. Nowadays, more and more content, including educational material has been created and published on the web. With Web 2.0, individuals themselves become creators of web content like articles or blogs, comments on SNS, and “tweets” on twitter. Further, although the Internet is full of information, finding the right information can be difficult even with the most advanced search engines. And information is not necessarily knowledge.

The evolution of the Internet has provided users with different ways of finding information. Traditionally, users did not concern themselves with searching for information because there was not much available to find. However, as more and more websites have become available and as technology has moved on, search engines like Yahoo and Google have been developed and used to retrieve information. Search engines return lists of websites containing information relevant to the search criteria entered by the user. The user then needs to drill down into the returned information to find exactly what they are looking for. However, despite being full of information the Internet is not necessarily providing knowledge. To get closer to doing that, the information would need to be well-organized and well-presented, which means that web platforms must also be well-organized and well-

presented. As a result of this improved approach to organizing information, the problems of user distraction and information overload might also be resolved.

Some websites already serve as platforms for collecting, organizing and sharing information: examples are Wikipedia, Howstuffworks.com and Wolfram Alpha [124]; Microsoft Office templates [125]; and Matlab File Exchange [126]. Instead of digging for information through thousands of web pages returned by search engines, many users prefer to use these websites to find information directly. These websites are acting as repositories or knowledge bases. A knowledge base is a central location where knowledge is collected, organized, shared, searched, and utilized. Over the years, users have benefited from Wikipedia in terms of being able to access a basic understanding of new topics, determine search terms they want to use, and seek new ideas. In the same way, it can be very useful to have knowledgebase websites for specific fields or purposes, such as Wolfram Alpha for computer science data, Microsoft Office Templates for all kinds of business and administration templates, and Matlab File Exchange for exchanging Matlab scripts. Modelica language [127] has been developed as a unified language to describe physical systems on a computer. It has been used to develop many computer models of physical systems such as electrical circuits, car engines and complex systems spanning different domains. However, there is a lack of an easy-accessible knowledge base for these computer models, where they can be searched, shared, utilized and simulated.

3.2 Simulation Experiments and Internet Sharing

From Experiment to Simulation

Many undergraduate engineering courses involve concepts, principles and phenomena with which students are often unfamiliar. These can be difficult to explain using text and visual aids but can be better clarified and demonstrated using experiments in the laboratory. Experimental teaching methods have thus become increasingly popular in the teaching of engineering. Students who have access to laboratories can test new concepts and carry out experiments to determine how a principle works under different circumstances.

Learning through experiments is acknowledged to be one of the most effective ways of gaining new knowledge by many people including academics[128], teachers, and managers[129]. Kolb [130] proposed a 4-stage learning cycle: experiencing something by doing it (concrete experience), reviewing what happened (reflective observation), drawing conclusions about the experience by thinking further about it (abstract conceptualization), experimenting to try out the learning (active experimentation) and thereby forming new concrete knowledge. Of these 4 stages, doing experiments is the most critical stage for developing knowledge. Another theory from McCarthy [131] also states that the process of gaining new knowledge can be simplified by answering four key questions: “Why”, “What”, “How”, and “What if”. There are four stages to the process:

- Why: have some doubts or be curious about a new system
- What: be exposed to a lot of information about this new system - theories, principles and ideas
- How: start wondering how the new information applies to the system

- What if: continue to explore the system by doing some experiments, for example, what happens if this part is removed?

With both Kolb and McCarthy's theories, performing an experiment is the final and most critical stage to acquiring knowledge. Shen et al. [132] and Chen [133] also state that the importance of using experiments for teaching in the engineering field is especially significant.

Nevertheless, it is not always possible to access laboratory equipment. Outside the laboratory, computer simulation presents an effective alternative for active experimentation of ideas. Winsberg [134] notes that many discussions on computer simulation refer only to numerical experiment. Simulation involves applying scientific theories and doing data analysis, and many simulation tools provide visualization tools for observing phenomena and data analysis. These are all characteristics of experiments. Finkelstein et al. [135] demonstrate that using simulation is a more productive way of developing student conceptual understanding than physical experiments. Jimoyiannis and Komis [136] note that working with computer simulation helps students overcome their cognitive constraints and effectively apply the concepts learned.

Innovations in computer technology have produced many simulation programs that make it possible for users to perform experiments in this way using a computer. Such programs include Simulink, Dymola, Solidworks, Adams, Cosmos, Phet and WorkingModels. Wieman et al.[128] discuss using the simulation tool Phet to enhance learning in the

classroom. Simulation is used in diverse areas such as performance optimization, safety engineering, testing, education, video gaming, and in other situations where physical experiments would be too costly or too dangerous. Computer simulation therefore provides a cost-effective, fast and convenient way to perform physical experiments. However, most of the simulation programs like Simulink, Dymola, and Solidworks are commercial software and characterized by their large installation and memory consumption, and the need for a steep learning curve to understand them.

Internet Sharing

Users often try to solve the same problem over and again, not knowing that there is already a solution available on the Internet. In 1996, Fishwick [5] hinted that the Internet would become the future platform for publishing and sharing digital content. Today's Internet provides a great capacity for sharing and publishing digital content; in fact almost anything can be shared. People can use Flickr for sharing photos, YouTube for sharing video and Instructables [137] for sharing personal projects such as painting and homemade items. Despite this, however, there are very few simulation programs that provide the option to share simulation experiments over the Internet. Furthermore, there are only a few websites (web applications) currently available where users can share a model for others to try out, for example, Matlab File Exchange and Wolfram Alpha.

Even with these exceptions (Matlab File Exchange and Wolfram Alpha), information is not represented intuitively and there is little opportunity for user interaction. So, if using a solution from Matlab File Exchange, users will first need to install the correct version of

the Matlab software onto a local computer, which might be costly; they then need to download the Matlab code and figure out how to use it.

3.3 Information Representation for Playing

With the rise in cloud computing, many researchers have been drawn to explore the logical and intuitive way in which a website presents information to the user [122] and the format in which that information is presented. Traditional websites use text, pictures, animation, tables, equations and video clips. Information from Wikipedia and howstuffworks, for example, is presented in the form of text, links, video clips, and both static and animated images. Users can only interpret information, however, through reading, watching, and listening; they cannot play with it. Thus, the potential for interaction and learning through playing has been limited by this constraint.

Matlab File Exchange presents its content in the form of downloadable scripts. To use a script in Matlab File Exchange, users need to have access to a computer with the correct version of the software already installed. They also need to possess sufficient knowledge to understand the downloaded scripts. A double pendulum example on the Matlab File

Exchange website is used to illustrate these points:

Figure 3.1. The Animated Double Pendulum Model in Matlab File Exchange

As shown in Figure 3.1, Matlab File Exchange provides a thumbnail to represent the double pendulum model and uses separate sections for displaying information, tags, comments and ratings, and for downloading scripts. Typically, the site is used to download the required scripts, try them out on a computer offline and return to the site to leave comments and rate the process. Users will have no idea what the system is capable of doing before downloading, installing and running it on a local computer. However, the download-install-run process is time consuming and not always successful.

Another example, Wolfram Alpha, uses a language-based approach with a computational mechanism to convey information using equations, text, and images. As with Matlab File Exchange, the example of a double pendulum is used for illustration:

Figure 3.2. The Double Pendulum Physical System in Wolfram Alpha

Figure 3.2 shows a schematic view of what a pendulum system looks like followed by properties (2 Degree of Freedom), and then the equations that are used to calculate the angles and velocities of different joints. However, there is not much opportunity for user interaction in this model.

In conclusion, these solutions do not offer an effective means of representing computer models of physical systems in a knowledgebase web platform, either because they are not intuitive or because they lack the necessary opportunities for interaction.

In the proposed solution, computer models represent the subject in its physical form. For example, the suspension system of a car is assembled using small parts such as coil springs, shock absorbers and linkage arms. In the proposed platform, the spring, shock absorber

and linkage are separate components which can be connected together to form a model, and this model looks like the real suspension system. Moreover, users can interact with the model in an intuitive way, changing parameters or modifying individual components, and all carried out within a browser. Most importantly, users are able to see the result of their modifications straightaway.

3.4 Proposed Solution

A web platform called Proteus has been proposed and developed. This new platform has been designed for education and academic research, and is free to use. It provides a place where students, educators and academic researchers can easily create and share their computer models of physical systems. These systems have been described using Modelica, a non-proprietary, object-oriented, equation-based language for physical system modeling.

Proteus[138] comes with a web-based, graphical modeling and simulation tool called ProteusGWT. ProteusGWT is web-based and uses an intuitive, graphical component-oriented approach to modeling physical systems. It also has the capability of spanning multiple domains including systems containing mechanical, hydraulic, thermal, control, electrical, electronic, electric power or process-oriented subcomponents. It synthesizes state-of-the-art web technologies (HTML5, GWT, and cloud computing) with computational methods for physical systems modeling and simulation to create a computing environment that is widely deployable and scalable. Anyone with a computer or browser-enabled device will be able to use it and contribute their computer models of

physical systems to this platform. As the platform grows, it could develop into an online interactive repository for all kinds of physical-system models. For example, a student might want to examine a complete computer model of a motorcycle, a refrigerator, a burglar alarm, or robot arm and learn about how they all work. He or she could run simulations, modify the models or create new designs and share them with others. Nothing quite like this is currently available on the Internet.

Some key features of the Proteus system are highlighted below:

- **Collective Intelligence:** Proteus, like Wikipedia, relies on the collective intelligence [3] of the public to contribute content and also to determine the quality of uploaded content.
- **Web-based, component-based modeling:** it offers a web-based, graphic component-based modeling and simulation environment for physical systems. This allows students and researchers to conduct engineering experiments from any location, any time.
- **Intuitive representation of experiment models:** Proteus gives users the opportunity to figure out what a system does by playing with it within a browser instead of passively reading a text and watching animated pictures. The models are built to resemble the physical objects they represent. By just looking at the model, users can immediately tell how to change parameters and adjust the components of the model.
- **Multi-domain modeling:** using Modelica language, Proteus is able to model and simulate physical systems spanning mechanical, hydraulic, thermal, control,

electrical, electronic, electric power or process-oriented subcomponents. A large number of commercial and open-source Modelica libraries are available from which users can select to model almost any physical system in the world.

- **Rich Interface Application:** With the power of HTML5 and the modern JavaScript engine, Proteus provides almost the same user experience as a desktop simulation application in terms of performance and usability.
- **Search Function:** it preserves a user's searching habit on popular search engines like Google and Yahoo by providing a single search box with an advanced search option.

Chapter 4

Design of Proteus - the Web Platform for Physical Systems Modeling

This chapter describes the design and development of the Proteus system. Proteus is a web-based modeling and simulation environment that enables students, researchers and public users to have an enjoyable and effective learning experience by sharing and playing with simulation models. In the Proteus environment, physical systems are modeled and represented the way they actually look in reality. For example, a mass-spring system and its corresponding computer model on the Proteus system are shown in Figure 4.1.

Figure 4.1. The Physical System and the Computer Model of a Mass-Spring System

As can be seen, individual blocks look the same as their physical objects - a mass and a spring in this example. Individual blocks can be easily assembled and taken apart at will,

and each block contains parameters and attributes that can be configured for different simulation purposes. In this way, Proteus is enabling a customized computer model of the physical object to be simulated, and the simulation result can then be viewed and analyzed in a convenient way. Most importantly, all of these operations are carried out within a browser.

Chapter 4 is structured as follows: a system overview is presented in section 4.1; a scenario describing the use of the Proteus system is given in section 4.2; the software design and development issues of individual subsystems are discussed in section 4.3; section 4.4 lists several contributions and potential uses of the Proteus system; and section 4.5 provides a conclusion.

4.1 System Overview

As introduced in Chapter 3, Proteus is a browser-based, server-side, rich internet application. The approach taken to its development has ensured Proteus' significant processing power, high availability, and ease of integration with existing server-side technologies such as database technology, scripting languages and email notification. Figure 4.2 illustrates the overall system architecture. As discussed in section 2.6.2, Proteus leverages the built-in support of cloud computing to handle performance and scalability issues. Both software and database are installed onto the servers in the cloud. No additional software or plugin is needed on the client machine or browser. Proteus has adopted the software as a service (SaaS) approach. Services include a web-based, graphic, component-

based modeling platform; an on-demand simulation service; a web-based visualization tool for viewing and analyzing simulation results; and a searchable model repository for user-created models. Anyone can use these services from any internet-enabled device - desktop computer, laptop, tablet PC, or smart phone - via any existing network connection standard like Ethernet (i.e., DSL, Cable DSL, LAN or WAN), Wi-Fi, Cellular broadband and Satellite-based Internet.

Figure 4.2. The System Architecture of Proteus

Figure 4.3 shows the main interface of the Proteus platform. Like normal websites, it consists of four regions: top, left, middle, and right. The top region consists of a search box, a three-button navigation menu, and a login section. The left region consists of a category tree and a tag cloud system. The right region shows recent activities on Proteus such as recently uploaded experiment models and comments. The middle region shows promoted

Proteus nodes in a grid layout according to specific promotion criteria, for example, ordered by most viewed or highest rated.

Figure 4.3. The Main Interface of the Proteus Web Platform

4.2 A Scenario of the Proteus System in Use

A Proteus node is a representation of an experiment that consists of a simulation model and the necessary information about that experiment. Proteus nodes are shown on the front page in a thumbnail grid layout. A Proteus node has attributes like thumbnail, modified

date, rating, and owner name. The details that constitute an experiment, such as experiment objectives, parameters setting, and experiment result, and an online simulation tool for trying out the experiment, can be viewed by clicking into the corresponding thumbnail.

Users can comment and rate the experiment after trying it out. The original intention [52] of Modelica language was to create a unified modeling language for reusing and sharing simulation models. Reusing and sharing are made easy on Proteus. Users can make a copy of a shared experiment and save it under their own account for future modification and for extending the experiment. Figure 4.4 presents a scenario of Proteus in use.

Figure 4.4. A Scenario of the Proteus System in Use

This scenario includes four stages: view the experiment information; try out the experiment; leave ratings and comments; and make a copy and start working on it. Information in Proteus is presented logically and intuitively. For example, experiment information is organized as: physical system information, experiment objectives, experiment parameters setting, simulation setting, and experiment result. Taking a double pendulum system as an example, the information section of the experiment introduction, as shown in Figure 4.5,

shows a picture of a double pendulum to give users a general idea of what the system looks like. Other information is presented sequentially.

Figure 4.5. A Double Pendulum Physical System

“Try out the experiment” [130] and “make a copy and start working on it” are the stages that offer the best way of acquiring knowledge, as discussed. As shown in Figure 4.6, a gateway called “Edit in ProteusGWT” is shown on the same page as the experiment information and provides access to the web-based modeling and simulation tool, ProteusGWT. Users can see the double pendulum computer model in the ProteusGWT editor (as shown in Figure 4.7) after clicking the link.

Figure 4.6. Trying out the Experiment in the Web-based Simulation Tool

Figure 4.7 Playing with the Double Pendulum Model in the Web-based Simulation Tool

4.3 Software Structure and Subsystems of the Proteus System

The Proteus web platform can be divided into 6 sub-systems: a search engine system, a comments and rating system, a category tag system, a permission control system, a file upload system and an online simulation tool (i.e. ProteusGWT). Figure 4.8 depicts the software architecture of the Proteus system.

Figure 4.8. Proteus Software Architecture

Proteus has been designed based on a popular content management system (CMS) called Drupal [139] which is one of the best open-source content management platforms available. It is built, used and supported by an active and diverse community of users around the world.

4.3.1 Search Engine on the Proteus System

Proteus provides two ways to locate a model on the platform to start experimenting with: the search function and the category and tag system. Use of categories and tags to browse models on a website is expected. However, with the increasing popularity of search engines like Google and Yahoo, more and more users prefer to use search engines to find website

information. Although users might not always be sure about the best “search term” to use (e.g. Resistor), they can learn about “terms” in various ways including using a tag cloud system, using search engine recommendations or by making several attempts to search using similar terms. Nevertheless, search engines are becoming standard features on a website. It is normal practice for users to look for a search function when needing to find information on a website, especially on website with user-generated content such as Proteus. A search engine must be smart enough to understand users’ needs and then propose recommendations. Furthermore, for websites with user-generated-content, a search engine must be able to search the latest content uploaded onto the system, and this requires constant re-indexing (see section 4.3.1.2).

4.3.1.1 Search Engine Options

Heng [140] summarizes 3 ways to provide search engines on a website: the first is to install a search engine script; the second is to use a third party hosted search engine service, and the third is to use a custom search engine from a leading search engine company (e.g., Google custom search). Compared with the other options, building a search engine from scratch is more time consuming but it also offers greater flexibility on a user-generated-content website like Proteus. The disadvantages of the last two methods mentioned are:

- **Advertisement:** free services impose advertisements on the results pages.
- **Flexibility:** most services allow the search results pages to be customized, but many do not provide the kinds of facilities a user might want for controlling the output.

- **Re-Indexing Issue:** the frequency of re-indexing one's site might be limited because re-indexing by a third-party website generates bandwidth usage and security issues.

4.3.1.2 Discussions

There are several things to consider when installing a website search engine: indexing, re-indexing, frequency of re-indexing and full-text indexing. It is generally agreed that website content needs to be indexed so that it can be used by a search engine. Newly generated content is not searchable until it has been re-indexed. Re-indexing frequency depends on how often a website is updated. However, like any other computer operation, re-indexing consumes computing resources (e.g., CPU time, memory, and disk I/O). Thus, re-indexing frequency should be kept to the minimum. Furthermore, this issue should be counterbalanced when designing the re-indexing algorithm of a search engine. If the website content is updated only monthly, for example, then on-demand re-indexing is a reasonable choice. Whenever the website is updated, a re-indexing algorithm is invoked. Large Web 2.0 websites like Proteus, however, expect to have hundreds of updates in a short time frame. If on-demand re-indexing is used, the underlying operating system has to perform hundreds of re-indexing operations at the same time and this is not practical because each operation would probably take minutes to hours to complete. Therefore, re-indexing for the Proteus system needs to be performed according to a predefined schedule. Depending on the actual usage of the system, one can design an algorithm for this schedule.

Another issue concerns full-text indexing. It is clear that full-text indexing takes longer but gives more sophisticated search results than keyword indexing. However the price for this sophistication, as well as time, is that specific algorithms or techniques have to be used, as when indexing user uploaded files like PDF, Microsoft Excel and Word files. For a static website, it is enough to index keywords such as titles, categories and tags. For a Web 2.0 website, however, full-text indexing or more sophisticated indexing must be decided on a case-by-case basis.

4.3.1.3 A Proposed Approach

Drupal offers many extensions with which to build a search engine. The Proteus search engine is developed based on Drupal's default search engine extension. Keyword re-indexing is performed every half an hour; re-indexing is performed by means of the cron task in a Linux system. In Proteus, the cron maintenance task is executed by calling a website's cron page which is stored in the root directory. More details about setting a cron task in a Linux system can be found on [141]. Many website modules have periodic tasks to carry out that must be triggered by a cron maintenance task. This includes the search module (to build and update the index used for keyword searching), the aggregator module (to retrieve feeds from other sites), the ping module (to notify other sites about new or updated content), and the system modules (to perform routine maintenance and updating of system tables).

4.3.2 File Upload System

The file upload system is a customized module that has been developed to create experiments on the Proteus system. Drupal has rich APIs that allow module developers to create custom modules [142]. Figure 4.9 shows this module's interface:

Figure 4.9. The Web Interface of the Proteus Upload Module

As shown in Figure 4.9, this module has several fields: title, category, tags, information description, thumbnail upload field, and text model upload field (can accept a Modelica file and a zipped file). Computer models uploaded to the system are processed by the file pre-process module (as described in section 6.2.2). The result file of this module is a web-friendly JSON file (as described in details in section 6.5.1).

4.3.3 Comment and Rating System

One of the research issues raised by Fishwick [78] is the quality control of web published digital content (e.g., simulation models). As discussed in section 2.5.3, researchers have taken different sides on this issue. However, it is agreed that a way of determining the quality of published digital objects needs to be found. A comment and rating system is used to determine trusted digital objects by using the collective intelligence of internet users. Byrne et al. [3] state that collective intelligence is the creation of a knowledge base using the wisdom of crowds rather than the expert few. An example of using collective intelligence is Wikipedia. Proteus has adopted a comment and rating system which has been widely used on Web2.0 online shopping websites like Amazon and eBay to help users make buying decisions. Many internet users identify the quality of a product based on users' comments and the number of reviews. Since most internet users are already familiar with the online shopping process, a web-based model repository like Proteus can adopt this approach and make the model selection process akin to a model shopping process.

A comments and rating feature is essential for a model shopping system and needs to have:

- The ability to support nested comments.
- The ability to encourage users to leave a comment, perhaps by adding something like a points reward system.
- A sophisticated algorithm to organize comments and determine the order in which they are listed.

4.3.4 Category and Tag System

A category and tag system provides an easy way to browse and navigate between different types of models and concerns sorting and grouping information in an organized way: categories and tags are for users; not just for search engines.

A category system acts like a “table of contents” for a site and helps describe what the site is about. It should incorporate collected groups of information and be specific enough to help visitors understand the content. Categories in Proteus are based on the default categories in the Modelica Standard Library (MSL). MSL categories include electrical, fluid, magnetic, and mechanical categories. A category may be broken down into sub-categories, for example, the electrical category contains the sub-categories analog, digital and machines.

Tags are site keywords - micro-data or meta-data - and are used by search engines. Tags can also be considered as micro-categories. The Proteus tag system is presented in a tag cloud. A tag cloud [143] is a visual representation of a collection of tags on a website. Key tags are emphasized in bold type and are usually in a larger or different colored font, as shown in Figure 4.10.

Figure 4.10. The Tag Cloud Module in the Proteus System

4.3.5 Other Subsystems

4.3.5.1 Permission Control System

A permission control system is used to control the visibility of uploaded models on the Proteus platform. For permissions to view these models, Proteus uses the same options as Flickr [144]: public, protected and private. Models with public visibility can be viewed, duplicated, and downloaded by anyone; Models with private visibility can be viewed only by the owners themselves and don't appear in search engine results; protected models are available only to owners and their friends. An enhanced permission control system would

allow model owners to share models with friends in a designated group as with the circles feature in Google+.

4.3.5.2 The Web-based Modeling and Simulation tool

A web-based modeling and simulation tool, ProteusGWT, is presented in Chapter 5.

4.4 The Contributions

The goal of this project is to create a place where users can search, learn and share knowledge about physical systems using web-based tools to carry out simulation experiments. The Proteus system makes several contributions to this area of research:

1. It proposes an online model repository for physical systems.
2. It provides a new way of representing information on a website i.e., a simulation model that can accurately represent a physical system on a website.
3. It proposes a web-based, easy-to-use, lightweight, high-availability modeling and simulation tool called ProteusGWT.
4. It creates a standard for web-based computer models, i.e., the ModelicaJSON model described in section 6.5.

4.5 Conclusion

ProteusGWT is an online simulation tool that integrates seamlessly with the Proteus web platform. When a user finds an interesting model on Proteus, he/she can launch ProteusGWT to view the model and run a simulation of it by clicking a ProteusGWT gateway link. Owners of these models can then further revise them in ProteusGWT by clicking on the link and then saving the revised version onto the server by clicking on the save button. The Proteus platform has been tested by a group of 20 students who have used it to build, upload, share and then modify their models in the Proteus system.

In summary, the Proteus platform consists of features that facilitate experimentation by enabling models to be created with parameters that can be set before running the simulation and subsequently revised. There is no need to install any software before use because ProteusGWT is a web-based modeling and simulation tool and any models created using the tool can be saved onto the server. This facilitates sharing among users and encourages collaboration. A user can quickly create a composite model with the tool, run the simulation and then view the results. The client workstation does not need to be well-equipped or have a high specification because the simulation takes place on the server. Technically, this infrastructure can support any number of users to create, simulate and share Modelica simulation models at the same time.

Chapter 5

Design of the Web-based Modeling and Simulation Tool

ProteusGWT (<http://www.visualphysics.net/ProteusGWT>) is a web-based modeling and simulation tool. It enables users to access and reuse simulation models on the Proteus system without leaving their browsers. It provides a convenient graphic tool for users to explore ideas, verify principles and create computer-based simulation models of physical systems. Users simply play with visual blocks to create these models using basic computer operations like drag, drop, delete, move, draw connection lines, and rotate and flip.

ProteusGWT is designed to meet Rich Internet Application (RIA) requirements. As discussed in Chapter 2, a RIA is a web application that has the same features and usability of a traditional desktop application but also has many advantages such as:

1. Cost-free installation on a large number of computers; a compatible web browser is sufficient to run the application.
2. It needs little or no disk space on the client.
3. There's no charge for upgrading the application.

4. It integrates easily with other server-side web services, such as email, database repositories and searching facilities.
5. It provides cross-platform compatibility in the majority of cases.

In addition to these advantages, ProteusGWT has minimal start-up time and the ability to work offline as a standalone model editor.

5.1 Overview of the Structure of ProteusGWT

Web applications use the client-server approach in which the client-side is an application running within a browser. ProteusGWT is a client-server web application developed with Google Web Toolkit (GWT).

5.1.1 Software Design Pattern

As shown in Figure 5.1, a web application such as ProteusGWT can be decoupled into several tiers: presentation logic, business logic, and data logic [145]. A regular website usually only has one tier on the client side. However, as the computing capability of web browsers becomes increasingly powerful, so the client-side presentation and business logic become more sophisticated and complex. Decoupling a web application into different tiers is one of the most essential tasks in application design. Where successful and well-designed, decoupling a web application can achieve high performance and fast response times by running heavy logic on the server side and critical logic (i.e., the part needing a fast response) on the client side.

Figure 5.1. Decoupling of a Web Application

Decoupling is applied to the entire ProteusGWT system including its individual subsystems, one of which is its front-end. The front-end is based on an MVP design. MVP stands for Model-View-Presenter and is a derivative of MVC (Model-View-Controller). Models are business objects or classes; views are user interfaces; and presenters are underlying logic or algorithms for different views. In terms of differences, MVP is more conceptualized in models so that models can be of any kind. In this way, it saves a lot of effort and redundant coding especially when an update of a model's structure is available. In summary, ProteusGWT takes the MVP approach for two main reasons: decoupling the front-end application allows multiple developers to work simultaneously; and minimizes redundant work when a model's structure changes.

5.1.1.1 Model

The business objects (i.e., Java classes) in ProteusGWT are listed below.

- **ComponentCanvasItem**: contains all Modelica component fields and is used to represent a Modelica component in ProteusGWT.
- **CanvasItem**: a light version of a Modelica component containing fewer fields.

- Item: the lightest version of a Modelica component containing only the unique identifier and the component name.

CanvasItem is a class inherited from Item class and is itself inherited by ComponentCanvasItem class. A ComponentCanvasItem class is a software representation of a Modelica component (e.g., a resistor component) in ProteusGWT. It contains almost all the fields of a Modelica component plus the corresponding graphic information which is held on the ProteusGWT canvas. Information about a Modelica component comprises fields such as a unique identifier, component name, the parameters and attributes of a Modelica model, inner components, connection information, icon graphics, diagram graphics, and documentation. A full list of Modelica component fields is presented in section 6.5.

5.1.1.2 View

Views are UI (i.e., user interfaces) components of ProteusGWT, including tables, labels, menus, trees, buttons, textboxes, and canvas. Views are responsible for the layout of UI components and have no relationship with models (i.e., business objects). This means that a view does not know it is displaying a Modelica component, it simply knows [146] it has a textbox, a canvas, and 3 buttons that are organized in a predefined layout.

Views in ProteusGWT are:

- DiagramDesignView
- IconDesignView

- CodeView
- DocumentationView

DiagramDesignView is the workspace in which computer models are built and edited. IconDesignView is used to design or view a computer model's small icons. DocumentationView displays Modelica documentation; CodeView its text code. A view management module within the presentation layer handles switching between different views.

5.1.1.3 Presenter

A presenter contains algorithms for generating views and exchanging data with the server. As a general rule [146], every view in ProteusGWT has a presenter that is used to drive the view and handle the events sourced from the UI widgets within that view.

For ProteusGWT, presenters are:

- DiagramCanvasPresenter
- IconCanvasPresenter
- CodeviewPresenter
- DocumentationPresenter
- LibraryPanelPresenter
- MessagePanelPresenter

5.1.1.4 AppController

AppController is a module that serves three purposes: it connects all presenters together, it acts as a communication hub with the server, and it manages all internal communication events among modules.

5.1.2 Appearance of the Web Application

ProteusGWT is a web-based component-based modeling and simulation tool with a multi-domain modeling capability. As shown in Figure 5.2, the user interface of ProteusGWT consists of:

1. A Menu and an Action Toolbar: the menu contains all links to related functions whilst the action toolbar comprises shortcuts to any frequently used functions when creating, designing and simulating models.
2. A Component-based Library Window: Modelica models are represented in visual blocks and can be dragged and dropped onto the design windows.
3. Design Windows, consisting of four sub windows: Icon Design Window (IDW), Diagram Design Window (DDW), (Modelica) Code View Window (CVW), and Documentation View Window (DVW).
4. A Model Tree Window: it shows all components/blocks of the current DDW in a tree structure.
5. A Feedback Message Window: in software design, feedback is one of the key design criteria. Feedback messages give users an idea of what's going on, what might go wrong, and what precautions to take.

6. Many Dialogs: all dialogs in ProteusGWT are presented in Figure 5.3. Each dialog serves a different purpose such as create a new model, set simulation parameters, set component parameters and set component attributes.
7. A Simulation Result Visualization Window: in this window, users can select and plot variables for visualization and analysis purposes.

Figure 5.2. Overview of the ProteusGWT Platform

Figure 5.3. ProteusGWT Dialogs and Windows: New Model Dialog, Set Component Attribute Dialog, Set Component Parameter Dialog, Simulation Dialog, Simulation Result Plot Option Dialog, and Simulation Result Visualization Window

5.1.3 Internal Structure of ProteusGWT

ProteusGWT is based on the MVP design pattern. However, the actual design is customized as needed. The ProteusGWT front-end has three layers: view layer, presentation layer, and application controller layer. As shown in Figure 5.4, the ProteusGWT front-end is modularized into 3 layers and 6 modules (i.e., there are 6 modules in the presentation layer).

Figure 5.4. Overall Modularized Structure of the ProteusGWT Front-end

As shown in Figure 5.5, the ProteusGWT back-end consists of four engines: the Modelica file pre-process engine, the library pre-process engine, the model file loading engine and the simulation engine. The first two engines - Modelica file and library pre-process engines - are used by the file upload module of the Proteus web platform. The last two modules are used by ProteusGWT for visualization and simulation services.

Figure 5.5. Modularized structure of ProteusGWT back-end

5.2 ProteusGWT Toolbar with Standard Functions

A menu bar and an action toolbar are standard features of a desktop application and ProteusGWT, as a Rich Internet Application, also has these features. As shown in Figure 5.2, the menu bar contains shortcuts of all the actions that can be used with ProteusGWT. Commonly used actions such as “Create New”, “Save”, “Undo/Redo”, “Zoom in/out”, “Rotate”, “Flip”, “Simulate” and “Plot” are available on the toolbar below the menu panel for convenient access. A login function is also available on the menu bar for users who wish to sign in to save models directly to the Proteus platform. Below is the list of actions that are available in each field of the menu bar.

File menu: contains a list of commands that can be used to perform various actions such as create a new model, open an existing model, save (upload) models to the web platform, export models into different formats including Modelica file, PDF, and Image (JPEG, PNG, GIF), and print the diagram from *Diagram Design Window*.

Edit menu: consist of commands for actions such as undo and redo in *Diagram Design Window*, and delete components or connection lines in *Diagram Design Window*.

View Menu: consists of commands that one can use to perform actions such as zoom in and zoom out of *Diagram Design Window*.

Model Menu: consists of commands that can be used to perform actions such as clockwise rotate, anti-clockwise rotate, horizontal flip and vertical flip of blocks on *Diagram Design Window*.

Run Menu: includes commands for performing different actions such as simulate and plot simulation result.

5.3 ProteusGWT Library Visualization Module

5.3.1 Structure of the Library Panel

A library panel is used to create user defined Modelica models using a drag-and-drop technique. The ProteusGWT library panel can use Modelica libraries from different sources: Modelica Standard Library (MSL), other free and commercial libraries on the Modelica Association website, and also from user-defined libraries, as shown in Figure 5.6.

Figure 5.6. Proteus Library Composition

MSL and other free or commercial libraries can be downloaded from the Modelica Association website [147]. User-defined libraries allow users to create more complex

models by encapsulating existing models into small blocks and then saving them into a user-defined library for later use. By the time this thesis was being written, only MSL was parsed and saved into the Proteus libraries module. In the future, libraries will be added to Proteus libraries to facilitate the creation of more library-specific models.

Another feature of this module is its search function. As shown in Figure 5.7, it appears as a search box at the top of the ProteusGWT library panel. This search function can eliminate the tedious work of navigating through the library tree to find components. It can also help users determine the right component to use for a specific purpose. The search function searches through all components and related documents as well and is therefore able to provide a comprehensive set of useful information. Figure 5.8 shows the documents relating to the resistor component block.

Figure 5.7. A Searchable Component-based library

Figure 5.8. The Document View of a Resistor Block

5.3.2 Library Pre-Process Engine

The library pre-process engine is a key module of ProteusGWT as it provides basic components for the library panel. Modelica libraries downloaded from the Modelica website have first to be processed by this engine before they can be used on ProteusGWT. The pre-process operation separates essential information from an original Modelica model, for example, annotations, parameters and attributes. The details of this operation are in section 6.5. The main Modelica library of ProteusGWT is MSL 3.1. Components in MSL 3.1 are stable, of high quality and have been thoroughly tested. They belong to a diverse set of engineering domains including mechanical, electrical, and thermal. With ProteusGWT, easy-to-understand simulation models can be built using “blocks” drawn

from these libraries. “Blocks” are connected using a graphic wire and the finished simulation model is then translated automatically into Modelica language.

Each “block” (i.e. Resistor) in MSL 3.1 is a model written in Modelica language. In order to be used by a program, these model blocks need to be parsed and translated into a language that can be understood by the program (i.e. Java). A Modelica parser is used to parse and translate these model blocks in MSL. A Modelica parser is the core part of the library pre-process engine. More details on generating this parser are described in Chapter 6.

MSL is developed using an object-oriented language (i.e. Modelica), and the ProteusGWT programming environment (i.e. J2EE) is also object-oriented. This makes it simple and straightforward to map each Modelica model to a Java object. MSL library is organized in a tree structure and each tree node is a Modelica model associated with a Java object in ProteusGWT. When users double click or drag-and-drop a component (i.e. Resistor) from the library tree, the corresponding Java object is transferred from the server to the ProteusGWT front-end.

One disadvantage of this approach is that the server has to keep all MSL Java objects in memory at all time. Since there are 922 models and 615 functions in MSL 3.1 this makes memory usage rather inefficient. Another disadvantage of this approach is the length of time it takes from the click on a component (e.g., a double click on the resistor component in the ProteusGWT library panel) to the display of the component diagram on the canvas.

For this reason, JSON (JavaScript Object Notation) is used to represent Modelica models in ProteusGWT. As discussed in section 2.4.3, JSON is a text-based open standard for representing data structure. Figure 5.9 shows how the Modelica models are first parsed in MSL and then translated into Java objects. They are next converted into JSON Objects (via the Gson library) and finally, when a component is clicked, the corresponding JSON object is collected from the server by ProteusGWT. As noted in Figure 5.9, a MySQL database is created during the process of translation into Java objects. More details on this Modelica MySQL database are discussed in section 6.5.

Figure 5.9. MSL 3.1 Database

5.3.3 ProteusGWT Library Icon Loading Algorithm

Another challenge for this component-based library is the application startup time. Tree items in the ProteusGWT library are displayed as image icons of which there are about

1000 in total. It takes 2-3 minutes to load them all if the network speed is slow. To resolve this problem, ProteusGWT has adopted a flexible strategy: it loads only the component icons that are actually going to be used. There are two ways to select a library component from ProteusGWT library panel: either by typing the component name inside the search box or by manually clicking to expand the library tree. Both methods trigger the program to load the descendant image icons. The icon-loading algorithm of library components is shown in Figure 5.10. The public method `onDemandIconload ()` is invoked by the two ways described above.

```
public void onDemandIconLoad (TreeItem treeitem, TreeNode parentNode) {
    List<TreeNode> children = parentNode.children;
    for (TreeNode level1Child : children) {
        List<TreeNode> level2Children = level1Child.children;
        TreeItem parentTreeItem = level1Child.treeItem;
        preloadChilren(parentTreeItem, level1Child);
    }
}

private void preloadChilren(TreeItem parentTreeItem, TreeNode
parentNode) {
    List<TreeNode> children = parentNode.children;
    // set component
    if (children.size() == 0 && !parentNode.isPackage) {
        //load the component icon image from the server
        Icon icon = loadIcon();
        parentTreeItem.setWidget(icon);
        return;
    }

    // set package
    for (TreeNode child : children) {
        //load the child icon image from the server
        Icon icon = loadIcon();
        parentTreeItem.addItem(icon)
    }
}
```

Figure 5.10. The On-demand Image-loading Algorithm

5.4 The Myth of the Graphic Drag-and-Drop Canvas

ProteusGWT aims to provide a modeling and simulation RIA for the Proteus platform. The minimum requirement of ProteusGWT is a canvas to support 2D manipulation within a browser. Thanks to web technology developments there are now several ways in which an HTML canvas can be created to support 2D manipulation: direct use of HTML5 canvas with JavaScript; use of GWT third-party libraries such as gwt-g2d and Incubator; and use of GWT version 2.2 or later. Irrespective of the precise option chosen, the basic principle remains the same: JavaScript should be used to access APIs in HTML5 canvas.

By the time this tool was developed, HTML5 had only just been announced and there were not many powerful JavaScript libraries available in which to manipulate HTML5 canvas. However, it was not chosen for another reason. JavaScript is not like traditional programming languages (such as Java and C++). It was originally designed not as a powerful programming language but as a complementary tool for displaying HTML in a browser. JavaScript lacks the proper development and debugging tools that Eclipse/NetBeans has for Java and Visual Studio has for C++. However, JavaScript libraries and proper development and debugging tools have now started appearing on the market.

Another way to produce an HTML canvas is by using GWT. This combines the advantages of JavaScript and the powerful, convenient programming language of Java. GWT is a library developed by Google which can compile, automatically, large-scale Java applications into JavaScript web applications. GWT has supported HTML canvas since version 2.2. However, when this project was developed, GWT was still at version 2.1. Hence, a third-party library called gwt-g2d [148] was used to develop HTML canvas in ProteusGWT.

The HTML canvas developed with gwt-g2d supports comprehensive 2D manipulation such as drag-and-drop a component, rotate or flip an icon, manipulate image data, zoom in/out, draw graphic lines, rectangles, arcs, texts, etc. Other standard editing options (delete, undo/redo) are also provided. The scope of this thesis does not include a detailed description of a gwt-g2d implementation.

5.4.1 Annotation Structure of a Modelica Model

Modelica language has a feature for embedding graphic information in a Modelica model, enabling users whose programming knowledge is not extensive to create complex simulation models in a graphic way. Graphic information, called “Annotation” in Modelica, is used to specify how a component is visualized, for example, as an Icon or a Diagram. There is a formal specification for Modelica annotations (i.e. Modelica Specification version 3.1 Chapter 17). According to Modelica Specification 3.1 [149], annotations are intended for storing additional information about a model, such as graphical objects,

documentation, code generation, or versioning. In this section, the main focus is on graphic annotations.

Graphic annotation has been used to specify information for icons, diagrams, and connection lines. Graphic annotation is presented in layers. An example of a layer specification from Modelica Specification 3.1 is shown in Figure 5.11. The layer description follows Modelica grammar and lexical conventions. For example, a pair of double quotation marks (i.e. “”) means the enclosed element is a string, and a pair of square bracket (i.e. “[]”) means the enclosed section is optional. The icon layer and diagram layers have the same structure consisting of two sections: coordinate system specification and graphics specification.

Layer descriptions (start of syntactic description):

```
icon_layer :  
  "Icon" "(" [ coordsys_specification "," ] graphics ")"  
  
diagram_layer :  
  "Diagram" "(" [ coordsys_specification "," ] graphics ")"
```

[Example:

```
annotation (  
  Icon(coordinateSystem(extent={{-100,-100}, {100,100}}),  
    graphics={Rectangle(extent={{-100,-100}, {100,100}}),  
              Text({{-100,-100}, {100,100}}, textString="Icon")});  
]
```

Figure 5.11. A Description of Annotation (From Modelica Specification 3.1)

As mentioned earlier, Modelica is an object-oriented language and Modelica annotation is also object-oriented in its approach. For example, if a term “A” is followed by a bracket (e.g., “Icon” “(“ “)”), this term is considered as an object, and if a term is followed by an

equal sign “=”, then it can be considered an attribute. For example, Figure 5.12 shows the annotation section of a resistor component. The annotation object contains three objects; icon, diagram, and documentation. The icon object contains two objects; coordinate system and graphics. The graphics part of the icon contains three objects; a rectangle, a line and another line, which are basic graphic elements defined with attributes.

```

model Resistor "Ideal linear electrical resistor"
  annotation (
    Icon(coordinateSystem(
      preserveAspectRatio=true,
      extent={{-100,-100},{100,100}},
      grid={2,2}), graphics={
        Rectangle(
          extent={{-70,30},{70,-30}},
          lineColor={0,0,255},
          fillColor={255,255,255},
          fillPattern=FillPattern.Solid),
        Line(points={{-90,0},{-70,0}}, color={0,0,255}),
        Line(points={{70,0},{90,0}}, color={0,0,255}),
      Diagram(coordinateSystem(
        preserveAspectRatio=true,
        extent={{-100,-100},{100,100}},
        grid={2,2}), graphics={
          Rectangle(extent={{-70,30},{70,-30}}, lineColor={0,0,255}),
          Line(points={{-96,0},{-70,0}}, color={0,0,255}),
          Line(points={{70,0},{96,0}}, color={0,0,255})}),
        Documentation(info="<HTML> STRING </HTML>", revisions="<html>
        STRING </html>");
    end Resistor;

```

Figure 5.12. The Annotation Section of a Resistor Component

In Modelica language there are a total of 6 basic graphic elements (i.e. Line, Rectangle, Ellipse, Polygon, Text, and Bitmap) which contribute to the visualization capability of Modelica graphic objects such as icons and diagrams. These basic graphic elements are defined in a Modelica “record” type with their required attributes. “Record” is one of Modelica’s special classes. As shown in Figure 5.13, the Line element consists of attributes:

color, pattern, arrow, thickness, points, smooth type, and attributes inherited from the “GraphicItem” class. The rest of the elements definition can be found in appendix C.

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

Figure 5.13. Modelica Line Annotation Definition

Other Uses of Annotation

Annotation is also used to organize parameters as different tabs in the properties dialog of Modelica graphical editors. For example, some multi-body components have initialization, animation and advanced parameter tabs. Annotation is also used to enable or disable some variables in annotation graphics.

5.4.2 Drawing in ProteusGWT

As discussed in section 5.4.1, Modelica annotations (i.e., basic graphic elements) are parsed and mapped to Java classes for drawing in ProteusGWT. As shown in Figure 5.14, there is a corresponding Java class for each Modelica basic graphic element. The same hierarchy structure used in Modelica annotation is adopted for Java classes. For example, all basic elements are inherited from “GraphicItem” and “FilledShape” classes. A

GraphicCanvas class acts as a placeholder for all graphic items. An icon class has both a GraphicCanvas class and a coordinateSystem class, which corresponds to the annotation layer structure in Figure 5.11.

Figure 5.14. UML Class Diagram of the Graphic Drawing in ProteusGWT

As discussed at the beginning of this chapter, the basic elements in Java representation are used by a GWT graphic library (i.e. gwt-g2d) for drawing. The detail of each class is not covered in this thesis because it is already well illustrated in Modelica Specification 3.1 section 17.5.

The information presented above is fundamental to icon and diagram drawing on HTML canvas, but the actual drawing is far more complex because it involves hierarchy annotation lookup, coordinate system transformation, and connector identification.

5.4.3 Coordinate system transformation

The coordinate system defines where the graphic items (including the basic graphic elements and the composed graphic items such as icons, diagrams) are placed and also how they are placed in terms of rotation, orientation, and size.

A coordinate system in Modelica may appear in different forms. For example, some coordinate systems contain only one field, an attribute called “extent”, whilst others contain several fields such as “origin”, “rotation”, “grid” and “preserveAspectRatio”.

The “preserveAspectRatio” attribute defines the constraints imposed on the shape (i.e., icon) of a component. The “preserveAspectRatio” is a Boolean variable and is used when the icon is resized or scaled. The “grid” is a tool-dependent attribute and is used to specify how many grids this shape should occupy on a specific tool (e.g., Dymola). The key attribute is “extent” which may also contain information from other attributes such as “origin” and “rotation”. The early version of Modelica language had only the “extent” attribute for the Coordinate System.

The value of the extent attribute is defined using two points on a 2D coordinate system, i.e., $([x1, y1], [x2, y2])$. The default extent value is $([-100, -100], [100, 100])$ which defines the origin $(0, 0)$, the dimension $(\text{abs}(x2-x1), \text{abs}(y2-y1))$, and the flip information (i.e.,

$x_2 < x_1$ defines horizontal flipping and $y_2 < y_1$ defines vertical flipping around the point defined by the origin attribute).

A global coordinate system defines the actual dimensions of a graphic item when it is not used by any other component. The global coordinate system is defined and remains the same when a graphic item is defined. A placement (local) coordinate system defines the local position of a graphic item when it is used in another composed component. A placement coordinate system depends on the other composed component. Hence, when composing a complex graphic object (e.g., a resistor icon) with several small graphic objects, coordinate system transformation is required. A coordinate system transformation is for mapping the global coordinate system of each single item into its placement coordinate system in a composed object. An example is shown in Figure 5.15 where a resistor component contains a positive pin and a negative pin. The “extent” value for the positive and negative pin is $([-100,100], [100,-100])$, and their placement transformations in the resistor component are: $\text{extent} = ([-110,-10], [-90,10])$, $\text{rotation} = 0$ for the positive pin and $\text{extent} = ([110,-10], [90,10])$, $\text{rotation} = 0$ for the negative pin. When displaying the positive and negative pin within the resistor component, the extent value needs to be transformed according to the placement transformation value.


```

model Resistor "Ideal linear electrical resistor"
  PositivePin p annotation (
    Placement(transformation(
      extent={{-110,-10},{-90,10}}, rotation=0));
  NegativePin n annotation (
    Placement(transformation(
      extent={{110,-10},{90,10}}, rotation=0));
  annotation (
    Icon(coordinateSystem(
      preserveAspectRatio=true,
      extent={{-100,-100},{100,100}},
      grid={2,2}), graphics={
      Rectangle(
        extent={{-70,30},{70,-30}},
        lineColor={0,0,255},
        fillColor={255,255,255},
        fillPattern=FillPattern.Solid),
      Line(points={{-90,0},{-70,0}}, color={0,0,255}),
      Line(points={{70,0},{90,0}}, color={0,0,255}),
      Diagram(coordinateSystem(
        preserveAspectRatio=true,
        extent={{-100,-100},{100,100}},
        grid={2,2}), graphics={
        Rectangle(extent={{-70,30},{70,-30}}, lineColor={0,0,255}),
        Line(points={{-96,0},{-70,0}}, color={0,0,255}),
        Line(points={{70,0},{96,0}}, color={0,0,255})}),
    Documentation(info="<HTML> STRING </HTML>", revisions="<html> STRING
    </html>");
end Resistor;

connector PositivePin
  annotation(
    Icon(graphics={Rectangle(
      extent={{-100,100},{100,-100}})}));
end PositivePin;

connector PositivePin
  annotation(
    Icon(graphics={Rectangle(
      extent={{-100,100},{100,-100}})}));
end PositivePin;

```

Figure 5.15. An Example of the Modelica Resistor class which contains two small classes: A Positive Pin and a Negative Pin

Coordinate system transformation becomes more complex when there are attributes like “rotation” in the placement transformation section.

5.4.4 Flip and Rotation Manipulation on Canvas

Flip and rotation are standard user actions that apply to shapes on canvas. However, user actions to flip and rotate are often applied in an unpredictable order. To show the icon correctly according to a user’s actions, a state machine is often used with which to record the current state of a shape. The state machine used in ProteusGWT is shown in Figure 5.16. There are 8 states represented using A, B, C ... H. User actions such as clock-wise rotation, counter clock-wise rotation, flip horizontal and flip vertical are presented in acronym form CR, CCR, FH and FV respectively.

Figure 5.16. The Flip and Rotation State Machine for ProteusGWT Graphic Blocks

5.4.5 Response time optimization on the canvas

RIA encounters almost the same user experience as in desktop applications, making responsiveness one of its key criteria. ProteusGWT moves most of the presentation algorithms to the client-side in an attempt to increase responsiveness. Algorithms on the client-side are compiled into JavaScript by GWT. Therefore, the efficiency of JavaScript engine has a significant influence on responsiveness. Nowadays, the efficiency of a JavaScript program is lower than with a Java or C program. However, the difference is too

small to be observed in small web applications. Only in large web applications, especially ones that require large data manipulation, can the difference be seen.

One of the critical problems in ProteusGWT canvas is its performance. ProteusGWT canvas requires refreshing frequently in order to show a user's actions responsively. As already discussed, ProteusGWT canvas is actually an HTML5 canvas manipulated with JavaScript APIs. The drawing library (i.e. gwt-g2d) uses a double-buffered strategy, and the repaint is done manually (i.e., there is no timer to repaint the canvas automatically). The more shapes there are on canvas, the longer the repaint time. For instance, when the number of shapes increases to 50, the repaint takes 100~800ms depending on the complexity of each shape. In other words, when a user performs an action on the canvas, he/she has to wait 800ms to see the response. This does not constitute a high quality user experience.

The response time depends on several factors: the efficiency of a browser's JavaScript engine; the efficiency of the painting algorithm; and the number of shapes to repaint. One has little control over the first factor as it comes with the browser, however, it is recommended to use ProteusGWT with the Chrome browser because it has the fastest JavaScript engine. It is also possible to tune an application's repaint algorithm to improve performance. For example, one can reduce the nested loops and repeated calculations within the loops. The final repaint time for 50 shapes has reduced to 50ms~600ms after performing optimization on repaint algorithms.

The other approach is to reduce the number of shapes to repaint. ProteusGWT has adopted a Grid-Repaint algorithm, which divides the entire canvas into $4*n$ grids, where n can take from the pool of 1, 2, and 3. The canvas has an action boundary algorithm to determine on which grids the action happens and which grids have been affected. Only the affected grids need to be refreshed. This ‘refresh as required’ approach improves the performance dramatically and the repaint time of 50 shapes with 4 grids has dropped to 20ms~300ms. However, it requires careful design of the action boundary algorithm, especially when a shape covers several grids or a connection line between two shapes covers several grids. In consequence, the final version of this algorithm is not yet ready.

5.4.6 Connector Look-up

A Modelica connection statement contains a pair of connectors and is specified with annotations for visualization. Connector is a special type of Modelica class. As shown in Figure 5.17, the resistor component contains sub-components (i.e., connectors) “PositivePin p” and “NegativePin n”. Connectors are component interfaces used to make connections with other components. For example, a connection line between resistor r1’s positive pin and r2’s negative pin is established. The connection statement in Modelica is represented as “connect (r1.p, r2.n) annotation ();”. As one can see, the connector pin is denoted in a DOT notation (e.g., r1.p) and there could be multiple dots with multi-level hierarchies. The resistor example only has a one level hierarchy. As one can see from the example given, an annotation object follows the connect statement. The connection annotation is used to describe the properties of the connection line such as color, thinness, and pattern.

Besides that, as shown in Figure 5.17, each connector has its own annotation that defines the icon and placement. There is another type of connector called “expandable connector” which acts as a placeholder for all connections made to it. For example, there is a connection statement between a resistor *r1*’s positive pin and an expandable connector *RR*. The connection statement is “connect (*r1.p*, *RR.p*) annotation ()”. Whenever there is a connection to *RR*, *RR* will replicate the source’s connector (i.e. *r1.p*, *RR.p*). More details can be found in Modelica Specification 3.1. The connector detection algorithm (refer to section 6.4.3) is a complex and heavy duty task so it has been deployed on the server side.

```

model Resistor "Ideal linear electrical resistor"
  PositivePin p annotation (
    Placement(transformation(
      extent={{-110,-10},{-90,10}}, rotation=0));
  NegativePin n annotation (
    Placement(transformation(
      extent={{110,-10},{90,10}}, rotation=0));
  annotation (
    Icon(coordinateSystem(
      preserveAspectRatio=true,
      extent={{-100,-100},{100,100}},
      grid={2,2}), graphics={
      Rectangle(
        extent={{-70,30},{70,-30}},
        lineColor={0,0,255},
        fillColor={255,255,255},
        fillPattern=FillPattern.Solid),
      Line(points={{-90,0},{-70,0}}, color={0,0,255}),
      Line(points={{70,0},{90,0}}, color={0,0,255}),
      Diagram(coordinateSystem(
        preserveAspectRatio=true,
        extent={{-100,-100},{100,100}},
        grid={2,2}), graphics={
        Rectangle(extent={{-70,30},{70,-30}}, lineColor={0,0,255}),
        Line(points={{-96,0},{-70,0}}, color={0,0,255}),
        Line(points={{70,0},{96,0}}, color={0,0,255})})),
    Documentation(info="<HTML> STRING </HTML>", revisions="<html> STRING
    </html>");
end Resistor;

connector PositivePin
  annotation (
    Icon(graphics={Rectangle(
      extent={{-100,100},{100,-100}})}));
end PositivePin;

connector PositivePin
  annotation (
    Icon(graphics={Rectangle(
      extent={{-100,100},{100,-100}})}));
end PositivePin;

```

Figure 5.17. A Modelica Resistor Example for Connectors' Illustration

5.5 The Code Generator - Another Use of Modelica

Parser

The Code Generator module translates a graphic model in ProteusGWT canvas into a text model in Modelica language. Every change made to a component has to be closely translated. For example, resize, move, rotate, flip, add, delete, change properties on a component, and make connections from or to the component, and so on. The same Modelica parser as in the library pre-process engine is used to translate graphic models into text-based models. More details on the Modelica parser are discussed in Chapter 6.

5.6 The Simulation Process of ProteusGWT

The text-based model is sent to the server where it is compiled and simulated by an OpenModelica compiler. Figure 5.18 shows the simulation flow using ProteusGWT. As long as the simulation button on ProteusGWT is pressed, a piece of Modelica code (simulation model) along with simulation configurations will be sent to the server. To convert Modelica code into machine-code, ProteusGWT server uses an open-source symbolic translator (OpenModelica Compiler or OMC) to generate the corresponding C-code. This, in turn, is compiled into a binary executable using a compiler such as GCC on Linux or Visual C++ on Microsoft Windows. OMC is an open-source Modelica compiler supported by the Open-source Modelica Consortium. Once the simulation is complete, the ProteusGWT server returns the simulation result to the client which can then be plotted

using the client's plot module. The simulation result plotter is a 2D plotter for viewing the simulation result.

Figure 5.18. ProteusGWT Simulation Flow

5.7 Feedback - The Message Center

ProteusGWT is a rich Internet application (RIA) which means it has a feature-rich graphical user Interface (GUI) of which one of the key features is provision of timely and accurate feedback to users. As shown in region 5 of Figure 5.2, the feedback message center is found at the base of the ProteusGWT screen and consists of four individual windows: General, Info, Warning, and Error.

This message center follows the MVP design. There are four tabs: General, Info, Warning and Error. The text inside each tab is represented in a different color such as black for the General tab, green for the Info tab, yellow for the Warning tab and red for the Error tab.

When a message appears, the current tab view switches to the corresponding tab according to message type and appends the new message at the bottom. At the same time, the message panel scrolls down and shows the new messages to users. ProteusGWT supports editing of multiple models at the same time. Each model has its own message panel so that the information can be kept separate.

Most messages come from the ProteusGWT front-end and back-end. Messages from the ProteusGWT front-end are based on a user's actions on canvas, such as, add a new component or add a new connection line. Examples of messages from the back-end are simulation status, user login status and model save status.

5.8 Visualization of Simulation Result

The back-end simulation engine of ProteusGWT uses openModelica Compiler (OMC).

ProteusGWT uses commands to interact with OMC. The steps to simulate a Modelica model in OMC are shown as follows:

Step1:

loadModel(Modelica)

Step2:

loadFileInteractiveQualified("HelloWorld.mo")

Step3:

simulate(HelloWorld, startTime=0, stopTime=4, numberOfIntervals=10);

The first step is to load the required libraries such as Modelica Standard Library (MSL).

The second step is to load the Modelica model file, and the final step is to issue the simulation command which contains four attributes; name of the model, start time of the

simulation, stop time of the simulation and number of simulation intervals. As stated in section 1.3 of the OpenModelica User Guide [150], other attributes such as “outputInterval”, “method”(i.e., solving algorithms), “fixedStepSize”, “output Format” (i.e., the result file format), and tolerance are also available in the “simulate” command.

5.8.1 2D Visualization

A simulation result file contains by default the simulation data for every variable. As shown in Figure 5.19, the simulation result appears with both timestamp and value data. The left column is the timestamp and the right column is the value.

Timestamp	Value
0	1
4.440892098500626e-013	0.9999999999995559
0.4444444444444444	0.6411803884299349
0.8888888888888888	0.411112290507163
1.3333333333333333	0.2635971381157249
1.7777777777777778	0.1690133154060587
2.2222222222222222	0.1083680232218813
2.6666666666666667	0.06948345122279623
3.1111111111111112	0.04455142624447787
3.5555555555555556	0.02856550078454138
4	0.01831563888872685

Figure 5.19. A Sample of the Simulation Result of a Variable

The stream of the simulation result file is sent to the ProteusGWT client where it is plotted and analyzed with a visualization module inspired by Henrik Eriksson et al. [151]. An OpenModelica plot function and OMEdit function are examples of using this approach for 2D visualization of simulation result.

5.8.2 3D Visualization

There are several approaches to having 3D visualization support for a Modelica simulation. Eriksson [152] uses a method of embedding graphic objects into simulation models in his research with OpenModelica. He creates a new Modelica visualization package called Modelica.SimpleVisual. Dymola and SimulationX use a similar approach (i.e., graphical objects) for multi-body simulation.

From version 3.0, MSL contains a visualization package called Multibody Visualizer for 3D animation. Models from this package can be used like a normal Modelica package. For example, a Multibody model, “Body”, uses a cylinder and a sphere component from Multibody Visualizer as shown in Figure 5.20.

```

model Body
...
protected
    // Declarations for animation
    Visualizers.Advanced.Shape cylinder(
        shapeType="cylinder",
        color=cylinderColor,
        specularCoefficient=specularCoefficient,
        length=
        if Modelica.Math.Vectors.length(r_CM) > sphereDiameter/2
        then
            Modelica.Math.Vectors.length(r_CM) - (if cylinderDiameter > 1.1*
                sphereDiameter then sphereDiameter/2 else 0)
        else 0,
        width=cylinderDiameter,
        height=cylinderDiameter,
        lengthDirection=r_CM,
        widthDirection={0,1,0},
        r=frame_a.r_0,
        R=frame_a.R) if world.enableAnimation and animation;
    Visualizers.Advanced.Shape sphere(
        shapeType="sphere",
        color=sphereColor,
        specularCoefficient=specularCoefficient,
        length=sphereDiameter,
        width=sphereDiameter,
        height=sphereDiameter,
        lengthDirection={1,0,0},
        widthDirection={0,1,0},
        r_shape=r_CM - {1,0,0}*sphereDiameter/2,
        r=frame_a.r_0,
        R=frame_a.R) if world.enableAnimation and animation and
        sphereDiameter > 0;
end Body;

```

Figure 5.20. An example of using the Multibody Visualizer Package

Attributes of graphic objects (e.g., cylinder and sphere) are parsed and assigned with simulation values at every simulation step. These can then be used by other programs to create animations. Along the way, support from the compiler is needed to generate the corresponding coordinates of the annotation objects. By the time this thesis was being written, OpenModelica Compiler was not able to provide a simulation result of visualizer attributes for a number of models in the multi-body package.

Engelson et al. [153] propose another way to achieve 3D visualization by combining Modelica and SolidWorks models. A SolidWorks CAD model is first translated into standard Modelica notation which is then used for simulation and visualization. When a Modelica model is simulated, the position, orientation, velocity and acceleration of every element are computed. The computed data can then be associated with the SolidWorks model for dynamic visualization (3D animation). The design of ModelicaJSON is able to handle 3D visualization, but the 3D visualization of this tool is still under active development.

5.9 Applications

ProteusGWT uses an intuitive component-oriented approach for the modeling of physical systems. In this section, 3 examples are used to demonstrate: a two-mass two-spring system from a mechanical domain; a RC circuit system from an electrical domain; and a DCMotor system which uses both electrical and mechanical domains. The two-mass two-spring system, as shown in Figure 5.21, is a classic experiment in physics. There are two springs (one ideal and one damped) with spring constants k_1 and k_2 respectively and two masses m_1 and m_2 . One end of spring 1 is attached to a fixed wall; the other end is attached to m_1 while spring 2 connects m_1 and m_2 together.

Figure 5.21. A Two-mass Two-spring System

This model can be easily created using the component-based library and drag-and-drop canvas provided by ProteusGWT. In the component-based library, a user navigates to Modelica.Mechanical.Translational.Components and drags 2 units of mass components, 1 unit of spring component, 1 unit of fixed component, and 1 unit of springDamper component into the canvas on the right hand side. These then need to be arranged properly according to what the actual system looks like as in Figure 5.21. The user connects them together by clicking on the connector of each component. In the canvas, the connector is represented as a solid or empty rectangle depending on whether the connector is positive or negative. Figure 5.22 shows the complete model on the canvas.

Figure 5.22. The Complete Model of the Two-mass Two-spring System in ProteusGWT

Each component is associated with a property dialog. A right-click on the component in the drag-and-drop canvas and selection of the properties feature will bring up the property Dialog. The property Dialog is the place where parameters and initial conditions can be set before the simulation starts. For example, one can set the weight of both masses at 5kg, the spring constant of both springs at $1 \cdot 10^5$ N/m, the spring damping constant at $1 \cdot 10^2$ N.s/m, and the relative length of springDamper0 at 5m. Set the start simulation time to 0s,

the end simulation time as 5s, and the simulation interval as 500. When the simulation button is pressed, the ProteusGWT sends the generated Modelica code as well as the configuration values to the server. The server starts running the simulation and returns the result when complete. Once the simulation result is received from the server, the user can use the Plotter module to plot the graph (see Figure 5.23). In Figure 5.23, the displacement and velocity of mass m_1 and m_2 have been plotted.

Figure 5.23. Displacement for m_1 and m_2 in Plotter Module

Another example using ProteusGWT is from the electrical domain – an RC circuit. Figure 5.24 shows the completed graphic model of the RC circuit with a unit of constant power

supply, a unit of a resistor, and a unit of a capacitor. Set the value of the resistor at 1ohm, the capacitor at 1F, the constantvoltage at 1v and simulate 10s.

Figure 5.24. The Modelica model of an RC circuit

Once the simulation result is received from the server, the plotter module can be used to plot the voltage for the resistor and capacitor. The result is shown in Figure 5.25.

Figure 5.25. Voltage Plot of Simulation Result for the Resistor and Capacitor

The above two applications demonstrate the capability of Modelica when modeling physical systems from different domains but they do not show the capability of modeling multi-domain systems. A DCMotor example showing the multi-domain modeling capability, can be considered as an electrical circuit which also contains a mechanical component (Inertia) and a component from Blocks domain (Pulse), as shown in Figure 5.26.

Figure 5.26. An Example of Mixed Domain Modeling (DCMotor)

Set the proper parameters to what they are in Figure 5.26 and simulate for 100 seconds.

The angular velocity of inertia is plotted and shown in Figure 5.27.

Figure 5.27. Angular Velocity of Inertia

5.10 Conclusion

To sum up, ProteusGWT comes with a web-based, feature-rich, drag-and-drop canvas which is used to draw simulation models. Once the graphical model has been created, the corresponding Modelica file is then also created by the code generator. Thus, a user can create complex models and carry out simulations without needing to write Modelica code.

The views provided for Modelica models are important. There are four standard views: Icon view, Diagram view, Code view, and Document view. Icon view is where Modelica model icons are created and displayed; Diagram view is the default view mode which shows the internal graphic design of Modelica models; Code view shows a Modelica model in syntax-highlighted text. Mode; Document view is the information page of a Modelica model which can help users to explore, learn and use Modelica models. For instance, when users are not sure how to set parameters for a resistor block, they can refer to the document view of the resistor block.

Finally, ProteusGWT has a feature for exporting user designed graphic models into images, PDF or text models. It also provides a list of demos that beginners can use to learn modeling. Additionally, MSL provides domain-specific examples inside each package from which users can explore and learn.

Chapter 6

Design of the Modelica Parser and ModelicaJSON Standard

6.1 Introduction

As mentioned earlier, Modelica language has been developed to describe physical systems in computer models in the best possible way. Fritzson [154] notes object-oriented design, multi-domain, equation-based and a-causal modeling make Modelica superior to current modeling technologies. Modelica is a statically strong typed language[154] which allows this language to support standardized physical unit and is more suitable for safe modeling practices as the compiler can check the consistency of a design before it is executed. It has been developed in an international effort with the aim of creating a unified modeling language for complex physical systems spanning multi-domains. It is an open standard. There are many free and commercial Modelica libraries that are developed with it. Hence, Modelica has been selected as a reference for the development of the new web document format-ModelicaJSON. The new web document format which can represent simulation models also implements key modeling features of the Modelica language.

Although Modelica has no language elements to describe partial differential equations (PDE) directly, there are Modelica libraries (e.g., PDELib) and extensions[155-157] to address this issue. Nevertheless, Modelica is one of the most popular modeling languages used in many different fields of industry such as fluid systems, automotive applications, mechanical systems, thermo-fluid, and energy systems. In the automotive sector, many of the leading automotive OEMs including Ford, General Motors, Toyota, BMW and Daimler are all using Modelica.

In order to use Modelica language, a parser to make it accessible from other programming languages or tools is needed. For instance, it needs to be able to transform, query and manipulate from Java. As discussed in previous chapters, the Modelica parser module is frequently used by other Proteus modules such as the library pre-process engine, file pre-process engine, and code generator module. In general, a Modelica parser is essential for using and visualizing Modelica models on the web. In addition, it is needed for developing the new web document format, ModelicaJSON.

6.2 The Overall Architecture

6.2.1 The Big Picture

A computer model that has been described using Modelica language is called a Modelica model. To simulate a Modelica model, a Modelica compiler is needed. The usual approach taken by a Modelica compiler is to parse and convert Modelica code into C code, which is

then compiled into binary machine-code with a native C compiler, such as GCC on Linux and Visual C++ on Microsoft Windows.

As depicted in Figure 2.9, the process of developing a Modelica compiler can be divided into three stages. The first stage involves taking Modelica source code as input and performing lexical and semantic analysis to produce a flat Modelica model. This model is then passed to an equation optimizer to sort and eliminate redundant equations. During the final stage, the Code Generator takes the output from the equation optimizer module and generates the corresponding C code.

6.2.2 A Modelica Parser

The design and implementation of a full Modelica compiler is outside the scope of this thesis. The main purpose of this chapter is to describe the issues that arise when designing a Modelica Parser. Therefore, only the first stage (i.e., Lexical Analysis, Semantics Analysis, and Inheritance Flattening) of the design process is explored here. As discussed in previous chapters, the Modelica Parser has been used in three places; the file pre-process engine, library pre-process engine and code generator module.

- Library pre-process Engine: it is used to process models from Modelica libraries (e.g., MSL) into a web friendly format (i.e., ModelicaJSON) so that the models can be used in a web-based environment such as ProteusGWT.

- File pre-process Engine: is a light weight version of library pre-process engine in that it is used to process a single file in Modelica models into a web friendly format (i.e., ModelicaJSON).
- Code Generator Module: is used to translate a graphic model (i.e., a drawing on HTML5 canvas) into a model in Modelica language.

The parser accepts two types of inputs: Modelica libraries and single Modelica model files. If a Modelica model uses components from a Modelica library (e.g., MSL), the specified library itself must be parsed before parsing the Modelica model file. The parsed libraries are stored in a Modelica Database (i.e., MySQL), which is then used for name look-up when parsing the model.

Figure 6.1 shows a detailed view of Modelica Parser. A lexical analysis parser, built with Modelica grammar and JavaCC, is first used to parse Modelica source files and create an AST tree. This is then used for semantic analysis and flattening.

Figure 6.1. A Modelica Parser to Generate a Web Friendly JSON File

Modelica semantics is a set of rules used for translating clauses (including inheritance and modification), instances, and connections into partially flattened Modelica files. In the partially flattened Modelica file, the name look-up operations for import clauses, short classes, extend clauses, instantiated clauses, and parameters are completed. Here also, the annotations for icons, diagrams, include clauses, extend clauses, and connection clauses

are parsed and flattened. Overall, this process identifies and flattens the component connectors of a Modelica component.

The final output of this Parser module is a web friendly file format called ModelicaJSON. ModelicaJSON is described in JSON language (similar to XML). JSON has been recognized as a better approach than XML in terms of its loading speed and usability with JavaScript applications on the client side [158, 159]. As shown in Figure 6.2, an example from JSON official website[160] is used to demonstrate the differences between JSON and XML. Though there are some arguments regarding JSON lacks of some key features such as Schema and Namespace, JSON is still a superior choice in most situations for several reasons. For example, JSON is more compact and can be easily loaded in JavaScript. When data is encoded in JSON, the result size is typically smaller than an equivalent encoding in XML, mainly because XML has starting tags, closing tags, and header information. Hence, the same amount of data encoded in JSON takes smaller transmission time over the network. Data in JSON format can be natively loaded as JavaScript object while XML data requires specific XML parsers. Since, JSON is natively supported by JavaScript language, JSON parsing is generally faster than XML parsing.

```
{
  "glossary": {
    "title": "example glossary",
    "GlossDiv": {
      "title": "S",
      "GlossList": {
        "GlossEntry": {
          "ID": "SGML",
          "SortAs": "SGML",
          "GlossTerm": "Standard Generalized Markup Language",
          "Acronym": "SGML",
          "Abbrev": "ISO 8879:1986",
          "GlossDef": {
            "para": "A meta-markup language, used to create
markup languages such as DocBook.",
            "GlossSeeAlso": ["GML",
"XML"]
          },
          "GlossSee": "markup"
        }
      }
    }
  }
}
```

An example of JSON Format

```
<!DOCTYPE glossary PUBLIC "-//OASIS//DTD DocBook V3.1//EN">
<glossary><title>example glossary</title>
  <GlossDiv><title>S</title>
    <GlossList>
      <GlossEntry ID="SGML" SortAs="SGML">
        <GlossTerm>Standard Generalized Markup Language</GlossTerm>
        <Acronym>SGML</Acronym>
        <Abbrev>ISO 8879:1986</Abbrev>
        <GlossDef>
          <para>A meta-markup language, used to create markup
languages such as DocBook.</para>
          <GlossSeeAlso OtherTerm="GML">
            <GlossSeeAlso OtherTerm="XML">
          </GlossDef>
          <GlossSee OtherTerm="markup">
        </GlossEntry>
      </GlossList>
    </GlossDiv>
  </glossary>
```

The same text expressed in XML Format

Figure 6.2. A text expressed in both JSON and XML Format

To summarize, though XML is still the dominating standard for representing information on the web, JSON is extreme useful for web application where fast, compact and convenient serialization of data is required. The structure (Abstract Syntax) of a Modelica model is stored using ModelicaJSON (i.e., a Modelica model described in JSON language) and is easily accessible from other programming languages and easy to transform, query and manipulate.

6.2.3 An Example of using Modelica Parser

Figure 6.3 presents a Modelica model of a resistor. Proteus Modelica Parser takes this model and uses it to generate the corresponding ModelicaJSON file, which is then used by ProteusGWT for visualization (as shown in Figure 6.4). The objectives of Proteus Modelica Parser are to remove the import clauses and flatten the extend clauses, for example:

```
extends Modelica.Electrical.Analog.Interfaces.OnePort;  
extends Modelica.Electrical.Analog.Interfaces.ConditionalHeatPort(T =  
T_ref);
```

Icon graphics and diagram graphics are first parsed and flattened. Parameters, attributes and modifications are then identified and flattened and used in the properties dialog shown in Figure 6.4. As an example, parameter Resistance R has been identified as “real” type, with the unit as “Ohm” and comment as “Resistance”. The “real” type is one of the default Modelica types. A “real” type parameter has attributes such as start, value, quantity, displayUnit, min, max, fixed, nominal, and stateSelect. More details about Modelica default types are discussed in section 6.4.3.

```

model Resistor
  parameter Modelica.SIunits.Resistance R(start=1);
  parameter Modelica.SIunits.Temperature T_ref=300.15;
  parameter Modelica.SIunits.LinearTemperatureCoefficient alpha=0;
  extends Modelica.Electrical.Analog.Interfaces.OnePort;
  extends Modelica.Electrical.Analog.Interfaces.ConditionalHeatPort(T
= T_ref);
  Modelica.SIunits.Resistance R_actual;
equation
  assert((1 + alpha*(T_heatPort - T_ref)) >= Modelica.Constants.eps,
"Temperature outside scope of model!");
  R_actual = R*(1 + alpha*(T_heatPort - T_ref));
  v = R_actual*i;
  LossPower = v*i;

annotation (
  Icon(coordinateSystem(
    preserveAspectRatio=true,
    extent={{-100,-100},{100,100}},
    grid={2,2}), graphics={
    Rectangle(
      extent={{-70,30},{70,-30}},
      lineColor={0,0,255},
      fillColor={255,255,255},
      fillPattern=FillPattern.Solid),
    Line(points={{-90,0},{-70,0}}, color={0,0,255}),
    Line(points={{70,0},{90,0}}, color={0,0,255}),
    Diagram(coordinateSystem(
      preserveAspectRatio=true,
      extent={{-100,-100},{100,100}},
      grid={2,2}), graphics={
        Rectangle(extent={{-70,30},{70,-30}}, lineColor={0,0,255}),
        Line(points={{-96,0},{-70,0}}, color={0,0,255}),
        Line(points={{70,0},{96,0}}, color={0,0,255}})});
end Resistor;

```

Parameters

Icon Graphics

Diagram Graphics

Figure 6.3. The Modelica Source Code of a Resistor

Figure 6.4. The Graphic Representation of a Resistor Component in ProteusGWT

6.3 The Lexical Analyzer and Parser

Modelica is an advanced modeling language that has been developed to model physical systems. The lexical parser of Modelica language has been developed to:

- Extract annotations from Modelica models for visualization of Icons, diagrams, connectors, and connection lines.
- Extract information from included primitive parameters to allow users to set parameter values for a customized simulation.

- To translate ProteusGWT canvas actions into Modelica code. These actions include drawing a connection line, drag-and-drop a component, rotate, flip and resize a component.

In order to perform semantics analysis and hierarchy flattening, a lexical analyzer or parser is needed to connect Modelica models to the appropriate programming language.

6.3.1 Modelica Lexical Analyzer

The programming view of Modelica code is a string of characters delimited by whitespace. The first step that a Modelica lexical analyzer takes is to uncover the meaning of Modelica code and translate it into “tokens”. Tokens must be identified in a given file, but they must also be classified at the same time. Some tokens are easily recognized as keywords (e.g., replaceable, parameter and final). Other token categories include literals, punctuation and so on. A section (i.e., Lexical conventions) of Modelica Specification 3.1 [149] discusses patterns so that they can be classified. As mentioned, Proteus uses JavaCC to develop the parser. The essential information required to create a parser is the Lexical definition and Grammar, and both these are available in the Appendix. A lexical definition of Modelica language is presented in Appendix A; a Modelica grammar definition in Appendix B.

6.3.2 Modelica Grammar

Grammar is a set of rules that apply to a language. Grammar guides the author to write clearly so that the reader can understand the language. Like any other language, Modelica

has its own grammar, as shown in Figure 6.5. This grammar has been used to develop a Modelica parser. Lexical analysis is a process that breaks Modelica code into “words”, whilst grammatical analysis is a process of using these “words” to construct meaningful “sentences”. The sentences can describe definitions of Modelica types, instantiated clauses, parameters, equations clauses, and Modelica modifications clauses. An important reason for creating or processing a grammar definition is to avoid any possible ambiguity. Specifically, the parser needs to look two tokens ahead in order to resolve any ambiguities. It does this using an LL(k) grammar where, in this case, k equals 2.


```
class_definition :
  [ encapsulated ]
  [ partial ]
  ( class | model | record | block | [ expandable ] connector | type
  |
    package | function | operator | operator function )
  class_specifier

class_specifier :
  IDENT string_comment composition end IDENT
  | IDENT "=" base_prefix name [ array_subscripts ]
    [ class_modification ] comment
  | IDENT "=" enumeration "(" ( [enum_list] | ":" ) ")" comment
  | IDENT "=" der "(" name "," IDENT { "," IDENT } ")" comment
  | extends IDENT [ class_modification ] string_comment composition
    end IDENT

base_prefix :
  type_prefix

enum_list    : enumeration_literal { "," enumeration_literal }

enumeration_literal : IDENT comment

composition :
  element_list
  { public element_list |
    protected element_list |
    equation_section |
    algorithm_section
  }
  [ external [ language_specification ]
    [ external_function_call ] [ annotation ] ";" ]
  [ annotation ";" ]
```

Figure 6.5. A snapshot of a Modelica Grammar Specification

6.3.3 Modelica Tree Structure

While processing lexical tokens and matching them with Modelica grammar rules, JavaCC also has features for automatically generating a syntax tree that represents the underlying structure of the file being parsed. First of all, tree nodes of lexical tokens based on the Lexical specification are produced. The AST tree is then built by associating these nodes

with parents, child and sibling relationships. The abstract view of this AST tree is shown in Figure 6.6.

Figure 6.6. The Abstract View of a Modelica Model

A snapshot of the parser code is shown in Figure 6.7. Further details are given in Appendix B.

```

/*
*****
Stored Definition - Within
*****/
OMStoredDefinition stored_definition()#StoredDefinition:{
    boolean final_ = false;
    String s;
    OMClassDefinition clsdef;
}
{
    (((<WITHIN>{
        jjtThis.within = true;
    }
    (s = name(){
        jjtThis.name = s;
    }
    )?";")?((<FINAL>{
        final_ = true;
    }
    )?clsdef = class_definition(){
        clsdef.final_ = final_;
        jjtThis.classDefinitions.add(clsdef);
    }
    ";")*){
        return jjtThis;
    }
}
/*
*****
Class Definition
*****/
OMClassDefinition class_definition()#ClassDefinition:{
    Token opt = null, t;
}
{
    (<ENCAPSULATED>{
        jjtThis.encapsulated = true;
    }
    )?(<PARTIAL>{
        jjtThis.partial = true;
    }
    )?(t = <CLASS>
    | t = <MODEL>
    | t = <RECORD>
    | t = <BLOCK>
    | (opt = <EXPANDABLE>)?t = <CONNECTOR>
    | t = <TYPE>
    | t = <PACKAGE>
    | t = <FUNCTION>){
        jjtThis.restriction = (opt != null?opt.image:"")+t.image;
    }
    jjtThis.classSpecifier = class_specifier(){
        return jjtThis;
    }
}
}

```

Figure 6.7. A snapshot of the Proteus Modelica Parser

6.4 Modelica Semantics

After the abstract tree has been created, it has to be analyzed according to Modelica semantics. Modelica semantics are described in a natural language, and there are no tools currently available that can claim to realize the full semantics of a Modelica specification. This section discusses several key semantics offered by most of the tools.

It is assumed that all code being parsed is syntactically and semantically legal so that the need for syntax checking and validation has been excluded from this thesis. However, it is still necessary to do this in order to understand precisely what is implied by the Modelica code. For example, a set of semantics that allow exemplification of other components within a model has been implemented.

6.4.1 Scoping and Name Look-up

As with any programming language, a name (e.g., a variable) defined in a Modelica model has a scope within which it is visible. The “name scope” is the region of a code where the name has a meaning that corresponds to its intended use. The following section explains some Modelica language semantics which are significant for understanding the scoping rules of Modelica. Afterwards, the related algorithms used for scoping analysis are presented.

User-defined classes are built from predefined types, functions, etc. The predefined types in Modelica are Real, Integer, Boolean, and String. Modelica also has predefined functions such as `der()`, and `sin()`. All these predefined names are accessible from anywhere within a program, including encapsulated classes. Thus, the scope of predefined names is global. A set of scope rules in Modelica is summarized in Modelica Specification as follows:

- The scope of a top-level class definition is the entire program, except for those classes that are encapsulated.
- The scope of a local defined class constant covers the whole class where it is declared, including all nested classes that are not encapsulated.
- The scope of a declared variable or parameter is the enclosing class, excluding local nested classes.
- The scope of an iteration variable is the body of the for-loop where it is implicitly declared.

Name look-up algorithms are implemented using a basic knowledge of Modelica language scope and the semantics that are discussed in the next section. For example, in the library pre-process engine, the name for import clauses, short classes, extend clauses, instantiated clauses, and parameters are all resolved using these name-lookup algorithms.

Different name types (such as import clauses and short classes) have different name-lookup algorithms but are all based on the basic name-lookup algorithm presented in Modelica Specification.

6.4.2 The Basic Name Look-up Algorithm

As described in Chapter 2, Modelica language uses an object-oriented approach which makes it easy to implement a Modelica library in a hierarchy structure. Figure 6.8 shows an example of the hierarchy structure of a Modelica standard library. If using the “Mass” model in this library, it should be referred to as “Modelica.Mechanics.Translational.Components.Mass”. This is the composite name of the “Mass” component (i.e., composed using dot-notation). However, if used within the “Components” package, it can be simply referred to as “Mass”, and this is called a simple name.

```
package Modelica
...
  package Mechanics
    ...
      package Translational
        ...
          package Components
            ...
              model Mass
                ...
              end Mass;
            end Components;
          end Translational;
        end Mechanics;
      end Modelica;
```

Figure 6.8. The Tree Structure of Modelica Language

The terms “Simple Name” and “Composite Name” are used to describe the basic Name Look-up algorithm. The basic algorithms Simple Name Look-up and Composite Name Look-up from Modelica Specification are presented below.

Simple Name Look-up

A simple name look-up is performed in the ways described below:

- Look-up among the declared named elements (i.e., class definition and component declaration) of parent class including elements inherited from base-class (i.e., extend a class).
- Look-up among the import names of the qualified import statements of the class and not including import statements from base-class (i.e., import statement are not inherited). For example, if the import statement is “A.B.C”, then “C” is the name to check. If the import statement is “D=A.B.C”, then “D” is the name to check.
- Look-up among the public members of all import packages within the lexical scope.

Composite Name Look-up

A composite name (e.g., “A.B.C”) look-up is performed as follows:

- The first identifier (i.e., “A”) is looked up with *the Simple Name Look-up Algorithm*.
- If the identifier (i.e., “A”) denotes a component, the rest of the name (i.e., “B.C”) is looked up among the declared named elements of this component.
- If the identifier (i.e., “A”) denotes a class, that class is temporarily flattened with an empty environment. The rest of the name is looked up among the declared named elements of the temporary flattened class.

These are only the basic descriptions of the look-up algorithm. The actual implementation involves more rules and semantics which are described in Modelica Specification, for example, encapsulated elements, and inner and outer elements.

6.4.3 Modelica Semantics and Algorithms

6.4.3.1 Look-up Algorithm of Import Statements

The following rules and semantics from Modelica Specification apply to import clauses of a class:

- Import clauses are not inherited.
- Import clauses are not named elements of a class or package which means that import clauses cannot be changed by modifiers or re-declarations.
- The order of import-clauses does not matter.
- One can only import from packages, not from other kinds of classes, i.e., in import A.B.C; or import D=A.B.C; A.B must be a package.
- An imported package should always be referred to by its fully qualified name in the import clause.
- Multiple qualified import clauses may not have the same import name.

An imported statement name can take three forms, e.g., “import A.B.C”, “import D=A.B.C”, and “import A.B.C.*”. The look-up algorithm should be able to identify the

composite name “A.B.C” from these forms and then proceed with *the Composite Name Look-up Algorithm*.

6.4.3.2 Lookup Algorithm of Extend Statements

Modelica extend statement is used to inherit features from a base-class. According to the grammar definition of Modelica language, an extend statement can appear in two places: in the short class (see next section for definition of short class); and in normal extend clause. A special aspect to note is that the base-class is a primitive type (i.e., Real, String, Integer and Enum), and the look-up algorithm should be able to identify these primitive types.

The extend name look-up algorithm is described in this way:

1. Check if it is already resolved or a primitive type
2. Check if it is from import clauses
3. Apply the simple and composite lookup algorithm

6.4.3.3 Short Class Definition

A short class inherits elements from a base-class rather than defining the elements itself.

Example 1:

```
type MassFlowRate = Real (quantity="MassFlowRate", final unit="kg/s");
```

Example 2:

```
class MassFlowRate
  extends Real (quantity="MassFlowRate", final unit="kg/s");
end MassFlowRate;
```

Examples 1 and 2 above show two forms of short classes which are identified using the “=” and “extends” sign as highlighted in yellow.

6.4.3.4 Look-up Algorithm of Component-Clause Statement

Every declaration within a class has a type. Some declarations are Modelica primitive types (e.g., Real, Integer, and Boolean) and some are class names of another Modelica class. The former category is used to define parameters, constants and variables; the latter category is used to instantiate another Modelica class. The name look-up algorithm for the former category is described in the next section; the name look-up algorithm for the latter category is described below:

1. Check if it is already resolved or a primitive type
2. Start Looking within the partial flattened component (e.g., check declaration name, and check inner class)
3. Check from import classes
4. Check from the upper scope (i.e., parents)

6.4.3.5 Primitive Type Checking

Real, Integer, Boolean, String and Enum are primitive types that are defined in Modelica language. Other short classes can extend these primitive types to form new Modelica types. As these are the elements that form parameters, they need to be separated out from normal composite elements.

Type checking is the final stage of flattening, and the algorithm for this is as follows.

1. Ignore the private elements
2. Check if it is a type variable (includes primitive type and non-component type)
3. If it is a type variable, identify the type (i.e., real, Integer, Boolean, String, and Enum) and parse the attributes for each type variable.
4. Merge attribute modifiers for identified type variables

6.5 ModelicaJSON Standard Proposition

As discussed in section 6.2.2, a Modelica parser has been developed to translate a Modelica model in a programming language (i.e., Java) so that a Modelica model can be displayed as a graphic block on a web-based canvas. This then enables users to interact (e.g., set properties) with these graphic blocks and simulate a Modelica model using third-party compilers. To create a web friendly representation of a Modelica model, the Java object is translated into a new web friendly format called ModelicaJSON. Figure 6.9 depicts the procedures involved in this conversion.

Figure 6.9. A ModelicaJSON Generator

6.5.1 The Web Friendly Model Structure

This thesis proposes a web friendly and simplified Modelica standard called ModelicaJSON. All models on Proteus system are in the ModelicaJSON format with the aim and intention of becoming the new web document format for simulation models. A

model that follows the ModelicaJSON standard is called a ModelicaJSON model. The ModelicaJSON standard is developed based on the FMI standard as discussed in section 2.5.1.6, with some adjustments and enhancements to fit the purpose of a new web document format for simulation on the web. For example, ModelicaJSON uses JSON instead of XML for data representation for the reasons listed in section 6.2.2. The ModelicaJSON standard includes model equations into the same JSON file instead of separating them into another C-code file so that it can be used directly by a web application. The ModelicaJSON standard includes the graphic information of a model so that it can be visualized on a web application, while FMI uses an image file for model visualization. Last but not least, ModelicaJSON is more suitable to be processed directly by web applications since it is based on JSON.

The root-level element of a ModelicaJSON model is an object called ComponentDTO. This consists of 6 elements: icongraphics, diagramgraphics (for visualization), parameters list (to enable users to set up parameters), include and extend components (for displaying internal containing components), and connection list (for displaying the internal diagram connections). The root-level structure of ComponentDTO is shown in Figure 6.10

Figure 6.10. A Web Friendly Document Structure - ModelicaJSON

A ModelicaJSON model is not a fully flattened model. The hierarchy structure of include and extend components is preserved to enable a hierarchy representation of a model on ProteusGWT canvas. Figure 6.11 provides further details about this ModelicaJSON standard.

6.5.1.1 The Detailed Description of ModelicaJSON Standard

Figure 6.11. The Detailed Structure of a ModelicaJSON Model

Figure 6.11 depicts the detailed structure of the ModelicaJSON standard. Comparing Figure 6.11 with Figure 6.10, the center table “Component1” corresponds to the main element “ComponentDTO” in Figure 6.10, table “IconGraphics1” corresponds to the “iconGraphics” element, “DiagramGraphics1” corresponds to the “diagramGraphics” element, “IncludeComponent1” is the basic element s of “includeComponentList”, “ExtendComponent1” is the basic element of “extendComponentList” in Figure 6.10, “ComponentConnect1” is the basic element of “connectionList”, and “Parameter1” is the

basic element of “parameterList” in Figure 6.10. The attributes of each element are defined in the following tables. On the top level, the Component element consists of

Attributes	Description
FullName	The full path name of a Modelica model, which is also the type of this component (e.g., “Modelica.Electrical.Analog.Basic.Resistor”).
Name	This is the short name of this component. It usually takes the last word from the full name (e.g., ”Resistor”).
Restriction	It is the component type (e.g., class, package, record, connector, function, block, etc).
Comment	This is the comments to this component including file version, author, generation tool, generated time and date.

Table 6.1. Attributes of Component Element

Elements IconGraphics and DiagramGraphics only contain one attribute for the graphic information. That one attribute is “Annotation”. Annotation is a JSON string that has a hierarchy structure of graphic objects (see section 5.4 for more details on this graphic structure)

Element IncludeComponent defines the declared components within a Component. It is defined in Table 1 as:

Attributes	Description
Type	The full path name of the declared class (e.g., “Modelica.Electrical.Analog.Basic.Resistor”). See the “FullName” attribute in Table 1.
Name	The unique declaration name (e.g., “Modelica.Electrical.Analog.Basic.Resistor r1”, then “r1” is the name).
ArraySubscripts	The array value of the declaration if the declaration is in the array form (e.g., “Modelica.Electrical.Analog.Basic.Resistor r1[3]”, then “3” is the array value).
IfFlagName	Some declaration has a condition (e.g. “if useAnimation=true then Modelica.Electrical.Analog.Basic.Resistor r1”, the “useAnimation” is the ifFlagName).
IsProtected	Defines the visibility of the element, default visibility is public. So if the element is specified as protected, this attribute will be set to true.
IsFinal	If an element is defined as final, then this element cannot be modified by a modification or by a redeclaration. This attribute is typically useful when flattening the class.
OuterInner	The definition from Modelica specification is “An element declared with the prefix outer reference an element instance with the same name but using the prefix inner which is nearest in the enclosing instance hierarchy of the outer element declaration.
Variability	Defines when the value can vary over time. It takes following values: constant, parameter, discrete-time and continuous-time.
Causality	Defines whether this element is input, output or both.
Extent	Extent, Origin and Rotation are 3 attributes to define the graphic location of this element. See section 5. 4.3.
Origin	See section 5.4.
Rotation	See section 5.4.

Table 6.2. Attributes of IncludeComponent Element

Attributes of ExtendComponent is a subset of IncludeComponent's attributes. Hence, it is not described here.

ComponentConnect is to contain the information for model equations. Element

ComponentConnect is defined as:

Attributes	Description
Annotation	It is a JSON "String" and has a hierarchy structure of graphic objects for drawing connection lines (see section 5.4 for more detail on this graphic structure).
StartPin	It is a composition of an element name and a connector name (e.g., r1.p, see section 5.4.6).
EndPin	Same as above.

Table 6.3. Attributes of Connection Element

Element Parameter is defined as follows:

Attributes	Description
Name	Same as in Table 2.
Value	Same as in Table 2.
PrimitiveType	It takes value from the 5 basic Modelica types: Real, Integer, Boolean, String and Enumeration.
Annotation	Same as in Table 2.
ifFlagName	Same as in Table 2.

ArraySubscript	Same as in Table 2.
isProtected	Same as in Table 2.
isFinal	Same as in Table 2.
OuterInner	Same as in Table 2.
Variability	Same as in Table 2.
Causality	Same as in Table 2.

Table 6.4. Attributes of Parameter Element

The 5 basic Modelica types:

Figure 6.12 The Structure of Modelica Pre-defined Types

Attributes	Description	Primitive Types
Name	The variable name.	Applied to all 5 primitive types
Start	The start attribute value when the simulation is started.	Applied to all 5 primitive types
Fixed	It is a Boolean attribute. If it is true, the attribute value is not changed during the simulation.	Applied to Integer, Real, and Boolean primitive types.
Value	The current value of this primitive type.	Applied to all 5 primitive types
min / max	The minimal and maximum value which this attribute can get in simulation.	Applied to Integer and Real types
quantity	The definition from Modelica Specification says, “quantity attribute are for base types Boolean, Integer, String and Real, in order to allow abstracted variables to refer to physical quantities (e.g., Boolean i(quantity=’Current’))”.	Applied for Boolean, Integer, String and Real primitive types
Unit	The unit of the attribute in simulation.	Real
displayUnit	The unit used for display on graphic tools.	Real
nominal	The nominal value for the variable.	Real
stateSelect	It is used to explicitly control state selection. see Modelica Specification 3.1, section 4.8.7.1.	Real

Table 6.5. Attributes of Pre-defined Modelica Types

6.5.1.2 Modification and Merging

Modification is another issue of flattening the Modelica hierarchy structure. According to Modelica Specification 3.1, Modification can happen at three places: variable declaration, short class declaration and extends-clauses. The modification expression contains modifiers which modify elements of the class (e.g., parameters). A modifier modifies one or more declarations from an inherited class by changing some aspects of the inherited declarations. The most common kind of modifier just changes the default value or start value of a parameter. For example, a Resistor component is declared as: `Modelica.Electrical.Analog.Basic.Resistor r1(R(start = 1) = 1.e-3)`. “`R(start = 1) = 1.e-3`” is the modifier which modifies the Resistor parameter “`R`”’s default value and start value. Figure 6.13 shows the common kind of modification for `IncludeComponent` and `extendComponent`.

Figure 6.13 The common kind of Modification for IncludeComponent and
ExtendComponent

A more advanced modification is to modify the type and/or the prefixes and possibly the dimension sizes of a declared element. This kind of modification is called a redeclaration. An element can only be redeclared if it is defined with the special prefix keyword-*replaceable*. However, the feature of supporting this advanced modification is not yet

available in current version of ModelicaJSON, neither are the modifications on short class and extend clauses. Merging of modifiers is an important process of flattening. Merging of modifiers means that outer modifiers override inner modifiers. The merging is hierarchical, and an element defined as final by the final prefix in an element modification or declaration cannot be modified by a modification or by a redeclaration.

6.5.2 Modelica Database

As discussed in section 6.2.2, a Modelica database was created during the translation from Modelica models into web friendly ModelicaJSON models so as to facilitate rapid searching and type checking of Modelica components,

As shown in Figure 6.14, a ModelicaDB is built to store models of Modelica libraries, including Modelica Standard Library (MSL) and other open-source as well as commercial Modelica libraries. The information stored inside ModelicaDB follows the structure of the ModelicaJSON model. With ModelicaDB and the Modelica parser, these web friendly Modelica libraries are available online and can be used freely within a browser (e.g., via ProteusGWT tool).

Figure 6.14. Modelica DB Structure

6.6 Conclusion

A Modelica model is presented using a simplified web friendly ModelicaJSON model has been shown in this Chapter. Though, this new web document format is developed based on Modelica modeling language, if a different physical system modeling language is used, the web document format won't be much different as long as it supports object-oriented

modeling, multi-domain modeling and a-causal modeling features. With the ModelicaJSON model, online visualization, graphic manipulation, and parameters customization are all made possible. This work can be extended in future with the study of additional subjects including:

- An implementation of the full Modelica semantics.
- The parsing and production of additional free and commercial libraries that can be used with the Proteus system.
- Make the ModelicaJSON standard compatible and convertible with the FMI standard, so that modeling and simulation environments that support or plan to support the FMI standard can also support ModelicaJSON standard.

Chapter 7

Use of Proteus web Platform for Hardware-In-the-Loop Simulation

In addition to the research that has already been discussed, this chapter focuses on the research conducted into the use of Modelica for hardware-in-loop (HIL) simulation. This simulation experiment has been designed to demonstrate the possibility of performing a Modelica HIL simulation on Proteus web platform. As outlined in section 2.5.5, hardware-in-loop (HIL) simulation is a tool used to replace some components of a software simulation model with the actual hardware. It is particularly useful for testing and validating hardware components and helps shorten the development cycles of electronically controlled mechanical systems. As discussed in section 2.5.5, a good deal of research [112, 161, 162] has been conducted into using Modelica for HIL simulation and this suggests that benefits could be obtained if ModelicaJSON, a web document format of Modelica language, were able to model the HIL system. However, much of the research into HIL simulation has been done with commercial Modelica simulators such as Dymola and SimulationX rather than with open-source Modelica simulators. Moreover, it is anticipated that users in the field of HIL simulation will want to share their models. The Proteus web platform, therefore, which already provides a sharing platform for simulation models of physical systems, will need to support the sharing of these HIL models.

7.1 Introduction

HIL simulation requires a piece of hardware to be connected to a client machine which can then be accessed by a software program. It would be beneficial if Proteus, as a sharing platform for simulation models of physical systems, could directly support hardware access on the client machine. In this way, users would be able to share, view, and play with HIL simulation models on the platform.

As discussed in section 2.5.5, a new Modelica device library, called `Modelica_DeviceDrivers` has been developed and published as an open source Modelica third-party library. The underline principle of this library is using extern C function[115] to interact with hardware drivers, which is exactly what has been used in this experiment. This library supports soft synchronization[116] which uses the same “when” mechanism that has been used here. Overall, this library is good to be integrated into the Proteus system, and it is also useful to utilize it in the ProteusGWT system to provide HIL modeling and simulation functionalities.

This chapter proposes a HIL experiment as a proof-of-concept into the possibility of supporting hardware access on the Proteus web platform. For security reasons, there are only a few web technologies (e.g., Java Applet and ActiveX) that have access to the hardware on a client machine. In this experiment, Java Applet is used to demonstrate the

HIL capability. A serial port is used to communicate with the external hardware device, which, in this case, is an Emant Bluetooth DAQ light sensor kit. A serial port, in contrast to a parallel port, is the physical interface through which data transfers in or out, a bit at a time. RS-232 has been a serial port standard for a very long time and has been used in applications such as industrial automation systems, scientific instruments and some industrial and consumer products. Modern computers without RS232 ports may require serial-to-USB converters to enable serial communication with external RS232 devices. A COM port is a software representation of a serial port in Windows. Each serial port is assigned with a unique COM port. Hence, choosing a serial port as a communication channel in this HIL experiment ensures a future-proof approach.

As shown in Figure 7.1, an Emant Bluetooth DAQ light sensor kit (from Emant Pte Ltd) consists of a Bluetooth sender/receiver, a data acquisition board with a light sensor and a battery pack. The Bluetooth sender/receiver is connected to the DAQ board via a serial port. The Bluetooth dongle on the left hand-side is connected to a USB port of a computer.

Figure 7.1. An Emant Bluetooth DAQ light sensor kit with a Bluetooth Dongle

Figure 7.2 shows the design and process of the HIL simulation experiment: a Hardware (Emant380) is connected to a client computer via a Bluetooth adaptor. A web application is launched to connect the hardware and the remote server. A Modelica simulation model is designed to read the Emant380 light sensor once a second. The simulation happens on the server-side.

Figure 7.2. A Real-Time HIL Simulation Setup

When the simulation is in progress, the process of running the Modelica model for simulation calls an external C function and the C function in turn communicates with the client-side web application (i.e., Java Applet) through the CORBA protocol. Once the data has been captured from the Emant Bluetooth DAQ, the Java Applet can pass the data to the server. In this sense, the server, the network, and the client are all participating in the simulation. Figure 7.3 shows a conceptual view of this simulation.

Two processes run concurrently on the server side: one is the CORBA communication process; the other is the Modelica simulation process. These two processes are connected using the inter-process communication protocol known as Named Pipes [104]. On the client-side, there are two communication modules: CORBA communication module and hardware communication module. CORBA module is responsible for communication with the server; the job of the hardware communication module is to exchange data with the external hardware which, in this case, is Emant380.

Figure 7.3. A Conceptual View of Real-Time HIL Simulation System

7.2 Simulation Experiment Setup

The way the simulation experiment is physically connected is shown in Figure 7.1. The simulation time is set to 9 seconds and within this time period the light sensor on Emant380 is covered up for the first 8 seconds and then exposed for the next 1 second. The real-time HIL model is simulated using OpenModelica compiler on a Windows XP server. The

simulation result is plotted with the “Plot(x)” function in OpenModelica Shell after the simulation has taken place.

As shown in Figure 7.4, the simulation procedure is described as follows:

1. Firstly, both server-side and client-side CORBA processes are initialized by an “init()” function.
2. The Java Applet client issues a connection request to the server via a “connect()” function.
3. The server creates an independent process and allocates the necessary resource for the client after receiving the connection request.
4. The server sends back an acknowledgement to the client.
5. After that, the Modelica simulation model starts to run on a Modelica process.
6. During the simulation, the Modelica process calls the CORBA process to get the sensor data from the client. The CORBA process then contacts the Java Applet client which reads the sensor data from the hardware.

Figure 7.4. The Schedule Diagram of Real-time HIL simulation

7.3 Simulation Result

Figure 7.5 shows the simulation result from the above experiment. As shown in Figure 7.5, there are three windows. The left window shows the output from the CORBA server process; the middle window shows output from the Java Applet on the client-side; and the right window shows the plotted simulation result after the simulation has taken place.

Figure 7.5. The Simulation Result of a Real-time HIL Simulation

7.4 Issues and Problems

Timing is a critical issue for HIL network simulation. In this system, there are three places where a delay can occur: at the server-side (execution time), on the network (transmission time), and at the client-side (processing time). Nowadays, computers are extremely fast; thus server execution time and client processing time are negligible. The network transmission time, known as network latency, is the part that generates the most critical delay. With the development of hardware equipment and the advent of Gigabyte network cards and fiber, for instance, the network latency on a LAN network has become negligible. Therefore HIL network simulation on a LAN network is no longer a problem. However, further research is needed into HIL simulation on a WAN network.

Time synchronization is another issue with HIL simulation. Time synchronization is where the clocks of several distributed computers are synchronized. In HIL simulation, a piece of hardware is connected to the simulation loop via another computer or hardware which is itself a computation device. Hence, HIL simulation can be considered as distributed computing. The clock on every computer needs to be constantly synchronized [163]. Even when initially set accurately, real clocks will differ over time due to ‘clock drift’. Some distributed simulations require the clock to be accurate to within milliseconds, whilst others only need to be accurate to within seconds. In our experiment, the time requirement is not so critical, so time synchronization is achieved using a “when” statement in an algorithm section of a Modelica model [104] :

```
when (time >= SimuNext) then  
  readSensorData();  
end when;
```

In the above syntax, time means the simulation time in OpenModelica and SimuNext means the real time. A simulation model exchanges data with the outside using the “extern” feature of Modelica language. This “extern” feature supports FORTRAN, C and Java programming languages. There are two processes running on the server side: one is the Modelica simulation process; the other is the CORBA server process. These two processes communicate using an inter process communication protocol called Named Pipes [104]. Named Pipes is designed to enable communication between a pipe server and pipe clients. Hence there are three components: a named pipe, a pipe server and several pipe clients. A name pipe is like a file that pipe server and clients can read/write data on a first-in, first-out basis. The implementation of Named Pipe is platform dependent. In this case, a Windows named pipe is created because the experiment machine is Windows-based. The named pipe server is created within the Modelica process; the named pipe clients are created within the CORBA server process. Each incoming client to the CORBA server has a corresponding named pipe client.

A further issue for consideration is the hardware interaction on the client-side. Emant380 hardware provides low-level programming APIs via Dynamic-Link Library (i.e., dll). Users can write drivers to read/write data from/to the Emant380 hardware in Microsoft programming languages such as VC, C++, and .NET VB. In order to use it with Java (e.g., Java Applet), Java Native Interface (JNI) needs to talk to the newly written Emant380 driver.

HIL simulation has many potential applications in the fields of engineering education and computer games. In engineering education, users can perform experiments by remotely controlling hardware in a laboratory. With the advancement of network technologies, massive multiplayer web games are being developed which allow players around the world to interact with each other.

7.5 Conclusion

A simulation experiment using Modelica for hardware-in-the-loop simulation on a normal Windows platform has been presented. This simulation experiment has been designed to demonstrate the possibility of performing a Modelica HIL simulation on Proteus web platform. It will be possible to extend this example and conduct more complex and time critical experiments in future. Another future work is to integrate Proteus system with the hardware interfacing drivers, Modelica_DeviceDrivers to support real HIL modeling and simulation on the platform.

Chapter 8

Conclusions and Future Work

This chapter provides a summary and conclusions of the principal ideas and arguments contained within the thesis. It also proposes several possible options for extending the research in future projects.

8.1 Summary

Learning through experimentation is acknowledged as being one of the most effective ways of acquiring and retaining knowledge and it is easy to see how computer simulation can be used as just such a means of experimentation. The Internet provides a powerful mechanism for publishing and sharing digital content, yet it does not offer significant web-based simulation and experimentation capability. The new standard, tool and platform we have developed and proposed in this thesis have been designed to deliver this capability: ModelicaJSON is the web document format; ProteusGWT is the web-based modeling and simulation tool; and Proteus is the knowledgebase web platform we have built for sharing the simulation models created using ProteusGWT and described using ModelicaJSON. Altogether, Proteus, as mentioned above and explained in Chapter 4 is a web-based modeling and simulation environment. In the web platform, simulation models have the same appearance as their physical systems. For example, a model of a car suspension system can easily be created using a spring block, a shock absorber block and a linkage

arm drawn from the component library. Since everything is handled intuitively, the usual barriers to modeling using this web sharing platform that users are often presented with, have been effectively reduced or eliminated.

A substantial amount of research has been conducted into the fields of web-based simulation and model repository, as discussed in Chapter 2. The focus of much of this research has been on the technologies concerned, such that web-based simulation has been technology rather than demand driven. Since web technology is such a fast evolving field, it is difficult to establish precisely which technology is the most suitable. Researchers into web-based simulation have now started to consider the users' perspective. Most users, for instance, prefer to use an intuitive graphic approach to building models; they prefer to avoid having to install plugins or migrate the data; and they expect applications to respond quickly.

The web-based modeling and simulation tool described in Chapter 5 provides users with sufficient opportunity to interact with the models by providing options to alter the parameters, the components or both. Using an object-oriented language such as Modelica, users can create their own models from a shared one using a reuse, enhance or adapt approach. The resulting simulation is available instantly from which they are then in a position to view the impact of their modification(s). As might be expected, a facility such as this results in faster learning whilst at the same time reducing costs.

Another important issue that has been considered is the performance and scalability of the system. With Rich Internet Applications (RIAs), there are two critical times when performance matters: application startup time and actions response time. Web application startup time is addressed in section 5.3.3; the response time of user actions is addressed in section 5.4.5. More advanced solutions for performance issues are offered by server technologies such as cloud computing, discussed in section 2.6.2, with techniques such as caching, mirroring, increasing the number of application servers and deployment in multiple locations.

However, there are still some issues that will need to be resolved before building such a system. For example, an appropriate modeling language is needed to describe the physical systems. Modelica is a non-proprietary, object-oriented, equation-based language for modeling complex physical systems spanning multiple domains, including systems containing mechanical, hydraulic, thermal, control, electrical, electronic, electric power or process-oriented subcomponents. FMI is a standard for model exchange by using a combination of XML files and C-code. Based on FMI and Modelica Specification, ModelicaJSON is a new web document format based on Modelica that have been defined and developed as part of this study. This web-friendly modeling standard is introduced in section 6.5.1 and can be used to describe simulation models of physical systems. It is capable of modeling physical systems combining both continuous and discrete behaviors as well as physical systems from multiple domains.

8.2 Future Work

In general, this research can be extended as follows:

- Further development of the knowledgebase web platform is needed. For example, a closer integration between the simulation environment and the knowledgebase web platform is needed to enable users to upload and share models directly from the simulation environment. More user-friendly features such as auto thumbnail creator are also needed.
- Further development of the Modelica parser to incorporate full Modelica semantics.
- The new web-friendly ModelicaJSON structure allows more physical system models to be created, tested and uploaded to the web platform. Furthermore, in addition to MSL, more libraries (i.e., MotorcycleDynamics, FuzzyControl) from the Modelica website can be supported to fulfill the many different requirements of creating models.
- The Modelica MySQL database of MSL as described in Chapter 6 can be used for further research, for example, to verify and check for correctness of translated models.
- Finally, and of equal importance, this platform can be extended to perform web-based hardware-in-the-loop simulation research which, although explored in Chapter 7, merits further investigation.

Reference

- [1] A. Taivalsaari, T. Mikkonen, D. Ingalls, and K. Palacz, "Web Browser as an Application Platform: The Lively Kernel Experience," Sun Microsystems Laboratories, U.S.A.2008.
- [2] netcraft.com. (2012, July 5). *July 2012 Web Server Survey*.
- [3] J. Byrne, H. Cathal, and P. J. Byrne, "A review of Web-based simulation and supporting tools," *Simulation Modeling Practice and Theory*, vol. 18, pp. 253-276, 2010.
- [4] H. M. Soliman and A. A. Alfantookh, "Web-Based Simulation: An Overview," 1998.
- [5] P. A. Fishwick, "Web-Based Simulation: Some Personal Observations," in *Proceedings of the 28th conference on Winter simulation*, 1996, pp. 772-779.
- [6] L. Whitman, B. Huff, and S. Palaniswamy, "Commercial Simulation Over the Web," in *Proceedings of the 1998 Winter Simulation Conference*, 1998, pp. -.
- [7] P. Lorenz, T. J. Schriber, H. Dorwarth, and K.-C. Ritter, "Towards a Web Based Simulation Environment," in *Proceedings of the 1997 Winter Simulation Conference*, Atlanta, Georgia, United States, 1997, pp. 1338-1344.
- [8] E. H. Page, R. L. M. Jr, and S. P. Griffin, "Web-Based Simulation In SimJava Using Remote Method Invocation," in *Proceeding of the 1997 Winter Simulation Conference*, 1997.
- [9] A. Buss, "Component based simulation modeling with Simkit," in *Simulation Conference, 2002. Proceedings of the Winter*, 2002, pp. 243-249 vol.1.
- [10] C.-C. Shen, "Discrete-Event Simulation On The Internet and the Web," in *Proceedings of the 1998 International Conference on Web-Based Modeling and Simulation*, San Diego, CA, 1998, pp. 57-62.
- [11] J. A. Miller, Y. Ge, and J. Tao, "Component-based simulation environments: JSIM as a case study using Java Beans," in *Proceedings of the 1998 Winter Simulation Conference*, Washington, D. C. US, 1998, pp. 373-382.
- [12] J. Hurwitz, "Sorting out middleware," *DBMS*, vol. 11, pp. 10-12, 1998.
- [13] E. Wolfgang, "Software Engineering and Middleware."
- [14] R. R. Kodali, "Asynchronous Web Service Calls Using WS-Addressing," *SOA & WOA: Article*, 2006.
- [15] IONA, "Using Orbacus version 4.3," 2007.
- [16] J. I. Rodriguez, J. M. Jimenez, F. J. F. Funes, and J. G. Jalon, "Dynamic Simulation of Multi-Body Systems on Internet Using CORBA, java and XML," *Multibody System Dynamics*, vol. 10, pp. 177-199, 2003.
- [17] C. R. Harrell and D. A. Hicks, "Simulation Software Component Architecture for Simulation-based Enterprise Applications," in *Proceedings of the 1998 Winter Simulation Conference*, Washington D. C., USA, 1998.
- [18] M. Aleksy, A. Korthaus, and M. Schader, *Implementing Distributed Systems with Java and CORBA*: Springer, 2005.
- [19] J. Wang, W. Lu, and W. Jia, "A Web-Based Environment for Virtual Laboratory with CORBA Technology," *International Journal of Computer Processing of Oriental Languages*, vol. 16, pp. 261-274, 2003.

-
- [20] R. Orfali and D. Harkey, *Client/Server Programming with Java and CORBA (2nd ed.)*. New York, USA: John Wiley & Sons, Inc, 1998.
 - [21] D. Box. (2000). *A Young Person's Guide to The Simple Object Access Protocol: SOAP Increases Interoperability Across Platforms and Languages*. Available: <http://msdn.microsoft.com/en-us/magazine/bb985060.aspx>
 - [22] A. Guru, P. Savory, and R. Williams, "A Web-based interface for storing and executing simulation models," in *Proceedings of 2000 Winter Simulation Conference*, Orlando, FL, USA, 2000.
 - [23] J. A. Miller, P. A. Fishwick, S. J. E. Taylor, P. Benjamin, and B. Szymanski, "Research and Commercial Opportunities in Web-Based Simulation," *Simulation Practice and Theory*, vol. 9, pp. 55-72, 2001.
 - [24] A. Levytskyy and J. H. Kerckhoffs, "Towards A Prototype Web-based Collaborative Simulation Environment," *SCS: paper of the 5th Euromedia Conference*, pp. 60-66, 2000.
 - [25] J. Farrell and G. S. Nezlek, "Rich Internet Applications The Next Stage of Application Development," in *Information Technology Interfaces, 2007. ITI 2007. 29th International Conference on*, 2007, pp. 413-418.
 - [26] C. F. Salisbury, S. D. Farr, and J. A. Moore, "Web-Based Simulation Visualization Using Java3D," in *Proceedings of the 1999 Winter Simulation Conference*, Phoenix, Arizona, United States, 1999, pp. 1425-1429.
 - [27] J. A. Miller, A. F. Seila, and X. Xiang, "The JSIM web-based simulation environment," *Future Generation computer system*, vol. 17, pp. 119-133, 2000.
 - [28] K. Perkins, W. Adams, M. Dubson, N. Finkelstein, S. Reid, C. Wieman, *et al.*, "PhET: Interactive simulations for teaching and learning physics," *The Physics Teacher*, vol. 44, pp. 18-23, 2006.
 - [29] C. O. Rourke, "A Look at Rich Internet Applications," *Oracle Magazine*, 2004.
 - [30] J. Byrne, C. Heavey, and P. J. Byrne, "SIMCT: An Application of Web Based Simulation," in *Proceedings of the 2006 Operational Research Society (UK) 3rd Simulation Workshop*, Royal Leamington Spa, UK, 2006.
 - [31] C. Cui and H. Ni, "Optimization Simulation of XML Performance Based on JSON," *Communications Technology*, vol. 8, pp. 108-114, 2009.
 - [32] M. Gonzalez and J. Cuadrado, "A Collaborative Simulation Environment For Multibody System Analysis," in *Product Engineering*, ed: Springer Netherlands, 2004, pp. 299-310.
 - [33] J. E. Lascano and E. P. d. Ejercito, "JPA implementations versus pure JDBC."
 - [34] R. E. Nance, "A history of discrete event simulation programming languages," *SIGPLAN Not.*, vol. 28, pp. 149-175, 1993.
 - [35] R. E. Nance and R. G. Sargent, "Perspectives on the Evolution of Simulation," *Oper. Res.*, vol. 50, pp. 161-172, 2002.
 - [36] Y. Lei, L. Song, W. Wang, and C. Jiang, "A MetaModel-Based Representation Method For Reusable Simulation Model," in *Proceedings of the 2007 Winter Simulation Conference*, 2007.
 - [37] D. Goldsman, R. E. Nance, and J. R. Wilson, "A brief history of simulation revisited," in *Simulation Conference (WSC), Proceedings of the 2010 Winter*, 2010, pp. 567-574.
 - [38] C. A. Roberts and Y. M. Dessouky, "An Overview of Object-Oriented Simulation," *SIMULATION*, vol. 70, pp. 359-368, June 1, 1998 1998.
 - [39] R. S. Nair, J. A. Miller, and Z. Zhang, "A Java-Based Query Driven Simulation Environment," in *Proceedings of the 1996 Winter Simulation Conference*, 1996.
 - [40] T. L. Veith, J. e. Kobza, and C. P. Koelling, "Netsim: Java-based simulation for the World Wide Web," *Computers & Operations Research*, vol. 26, pp. 607-621, 1999.

-
- [41] R. McNab and F. W. Howell, "Using Java for Discrete Event Simulation," in *Proceedings of the Twelfth UK Computer and Telecommunications Performance Engineering Workshop*, University of Edinburgh, UK, 1996, pp. 219-228.
 - [42] M. Rohl, F. Marquardt, and A. M. Uhrmacher, "Exploiting Web Service Techniques For Composing Simulation Models," in *Proceedings of the 2007 Winter Simulation Conference*, 2007.
 - [43] P. Fritzson, "Modelica A language for Equation Based Physical Modeling and High Performance Simulation," *Applied Parallel Computing Large Scale Scientific and Industrial Problems*, vol. 1541, pp. 149-160, 1998.
 - [44] S. Miller and T. MathWorks, "Modeling Physical Systems as Physical Networks with the Simscape Language."
 - [45] H. Elmqvist, "A Structured Model Language for Large Continuous Systems," PhD, Lund Institute of Technology, Sweden, 1978.
 - [46] A. Jeandel and F. Boundaud, "Physical System Modeling Languages: From Allan to Modelica," in *Proceedings of International Building Performance Simulation Association*, Prague, 1997.
 - [47] J. F. Broenink, "Modelling, simulation and analysis with 20-SIM," *Journal A*, vol. 38, pp. 22-25, 1997.
 - [48] H. Elmqvist and S. E. Mattsson, "An Introduction to The Physical Modeling Language Modelica," in *Proceedings of the 9th European Simulation Symposium*, Passau, Germany, 1997.
 - [49] P. Fritzson, *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*: IEEE Press, 2004.
 - [50] H. Elmqvist, M. Otter, and C. Schlegel, "Physical Modeling with Modelica and Dymola and Real-Time Simulation with Simulink and Real Time Workshop," in *Proceedings of Matlab User Conference*, 1997.
 - [51] H. Elmqvist, S. E. Mattsson, and M. Otter, "Modelica-The New Object-Oriented Modeling Language," in *Proceedings of the 12th European Simulation Multiconference*, Manchester, UK, 1998.
 - [52] P. Fritzson and V. Engelson, "Modelica - A Unified Object-Oriented Language for System Modeling and Simulation," in *Proceedings of the 12th European Conference on Object-Oriented Programming*, Brussels, Belgium, 1998.
 - [53] P. Fritzson and P. Bunus, "Modelica-A General Object-Oriented Language for Continuous and Discrete-Event System Modeling and Simulation," in *Proceedings of The 35th Annual Simulation Symposium*, 2002.
 - [54] (2000-2012, June 30). *Modelica and the Modelica Association*. Available: <https://modelica.org/>
 - [55] F. Donida, G. Ferretti, S. M. Savaresi, F. Schiavo, and M. Tanelli, "Motorcycle Dynamics Library in Modelica," in *Proceedings of the 5th International Modelica Conference 2006*, 2006.
 - [56] L. Liu and G. Frey, "Feasibility Analysis for Networked Control Systems by Simulation in Modelica," in *Proceedings of IEEE International Conference on Emerging Technologies and Factory Automation 2008*, 2008, pp. 729-732.
 - [57] C. Martin-Villalba, A. Urqui, and S. Dormido, "An approach to virtual-lab implementation using Modelica," *Mathematical & Computer Modelling of Dynamical Systems*, vol. 14, pp. 341-360, 2008.
 - [58] J. F. Broenink, "Object-oriented modeling with bond graphs and Modelica," in *International Conference on Bond Graph Modeling and Simulation*, San Francisco, CA, 1999.
 - [59] T. Walid, "A Core Language for Executable Models of Cyber Physical Systems," 2012.

-
- [60] P. A. Fishwick, "Using XML For Simulation Modeling," in *Proceedings of the 2002 Winter Simulation Conference*, 2002.
 - [61] A. pop, O. Johansson, and P. Fritzson, "AN INTEGRATED FRAMEWORK FOR MODEL-DRIVEN PRODUCT DESIGN AND DEVELOPMENT USING MODELICA," in *Proceedings of the 45th Conference on Simulation and Modeling*, Copenhagen, Denmark, 2004.
 - [62] MODELISAR, "Functional Mock-up Interface for Model Exchange," 2010.
 - [63] T. Blochwitz, M. Otter, M. Arnold, C. Bausch, C. Claub, H. Elmqvist, *et al.*, "The Functional Mockup Interface for Tool independent Exchange of Simulation Models," in *Proceedings of Modelica Conference 2011*, 2011.
 - [64] FMI. (2012, Nov 11). *FMI Support in Tools*. Available: <https://www.fmi-standard.org/tools>
 - [65] B. Thiele and D. Henriksson, "Using the Functional Mockup Interface as an Intermediate Format in AUTOSAR Software Component Development," in *Proceedings 8th Modelica Conference*, Dresden, Germany, 2011.
 - [66] CyDesign Labs Inc. (2012) Preview Release of Model-Based Design Optimization Platform. *Cloud-Based CyDesign Studio Platform to Be Demonstrated at CIMData PLM Road Map 2012*.
 - [67] B. Wire. (2012, CyDesign Labs, Inc. Announces the Acquisition of deltatheta UK Limited.
 - [68] DARPA, "DARPA FANG Challenge 1 (Mobility/Drivetrain) Guidelines," ed. United States: Defense Advanced Research Projects Agency, 2012.
 - [69] Z. Shi, S. Zhao, and S.-a. Zhu, "An Internet-based electrical engineering virtual lab: Using Modelica for unified modeling," in *Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference on*, 2011, pp. 555-559.
 - [70] O. Duarte, "UN-VirtualLab: A web simulation environment of OpenModelica models for educational purposes," in *Proceedings 8th Modelica Conference*, Dresden, Germany, 2011.
 - [71] M. Torabzadeh-Tari, Z. M. Hossain, and P. Fritzson, "OMWeb - Virtual Web-based Remote Laboratory for Modelica in Engineering Courses," in *Proceedings 8th Modelica Conference*, Dresden, Germany, 2011.
 - [72] E.-L. L. Sandelin, S. Monemar, P. Fritzson, and P. Bunus, "DrModelica A Web-Based Teaching Environment for Modelica," in *Proceedings of the 44th Scandinavian Conference on Simulation and Modeling*.
 - [73] Q. Liu, T. Xiong, Q. Liu, and L. Chen, "WebMWorks: A General Web-Based Modeling and Simulation Environment for Modelica," in *Proceedings of 9th International Modelica Conference*, Munich, Germany, 2012.
 - [74] P. Solin, O. Certik, and S. Regmi, "The FEMhub Project and Classroom Teaching of Numerical Methods," in *Proceedings of the 8th Python in Science Conference*, 2009.
 - [75] J. A. Joines and S. D. Roberts, "Fundamental of Object-Oriented Simulation," in *Proceedings of the 1998 Winter Simulation Conference*, 1998.
 - [76] M. Pidd, N. Oses, and R. J. Brooks, "Component-based simulation on the Web?," in *Proceedings of the 31st conference on Winter simulation: Simulation---a bridge to the future - Volume 2*, Phoenix, Arizona, United States, 1999, pp. 1438-1444.
 - [77] E. H. Page, A. Buss, P. A. Fishwick, K. J. Healy, R. E. Nance, and R. J. Paul, "Web-Based simulation: Revolution or Evolution," *ACM Transactions on Modeling and Computer Simulation*, vol. 10, pp. 3-17, 2000.
 - [78] P. A. Fishwick, "Issues with Web-Publishable Digital Objects," in *Proceedings of SPIE: Enabling Technologies for Simulation Science II*, 1998.

-
- [79] G. Iazeolla and A. D. Ambrogio, "A web-based Environment for the reuse of Simulation Models," 1998.
 - [80] J. L. Snoep and B. G. Olivier, "Java Web Simulation; a web based database of kinetic models," 2002.
 - [81] J. Akesson, M. Gafvert, and H. Tummescheit, "JModelica an Open Source Platform for Optimization of Modelica Models," in *Proceedings of the 6th Vienna International Conference on Mathematical Modelling*, TU Wien, Vienna, Austria, 2009.
 - [82] P. Fritzson, P. Aronsson, H. Lundvall, K. nNystrom, A. Pop, L. Saldamli, *et al.*, "OpenModelica - A Free Open-Source Environment for System Modeling, Simulation, and Teaching," in *IEEE International Symposium on Computer-Aided Control Systems Design*, Technische Universität München, Germany, 2006.
 - [83] J. Akesson, T. Ekman, and G. Hedin, "Development of a Modelica Compiler using JastAdd," *Electronic Notes in Theoretical Computer Science (ENTCS)*, 2008.
 - [84] G. Hedin and E. Magnusson, "JastAdd-a Java-based system for implementing frontends," 2000.
 - [85] J. Lundberg, "Getting Started with JavaCC," Institute of Mathematics and System Engineering, Vaxjo University, Vaxjo Swden, 2006.
 - [86] D. Broman, "Meta-Languages and Semantics for Equation-Based Modeling and Simulation," 2010.
 - [87] O. Johansson, A. Pop, and P. Fritzson, "ModelicaDB - A Tool for Searching, Analysing, Crossreferencing and Checking of Modelica Libraries," 2005.
 - [88] H. Elmqvist, D. Bruck, and M. Otter, "Dymola User's Manual Version 3.0," *Dynasim AB, Lund, Sweden*, 1996.
 - [89] P. Harman and M. Tiller, "Building Modelica Tools using the Modelica SDK," in *Proceedings 7th Modelica Conference*, Como, Italy, 2009.
 - [90] N.-G. Christoph, T. Ernst, A. Nordwig, and A. Mattes, "MOSILAB: Development of a Modelica based generic simulation tool supporting model structural dynamics," in *Proceedings of the 4th International Modelica Conference*, Fraunhofer Institutes, Germany, 2005.
 - [91] L. Allain, A. Merdassi, and L. Gerbaud, "Automatic modelling of Power Electronic Converter, Average model construction and Modelica model generation," in *Proceedings 7th Modelica Conference*, Como, Italy, 2009.
 - [92] Z. He, M. Xu, and F. Yang, "Multi-Domain Modeling Simulation and Analysis Using MapleSim," *Computer Knowledge and Technology*, vol. 36, pp. 66-75, 2009.
 - [93] F. Zhou, L. Chen, Y. Wu, J. Ding, J. Zhao, and Y. Zhang, "MWorks-a Modern IDE for Modeling and Simulation of Multi-domain Physical Systems Based on Modelica," in *Proceedings of International Modelica Conference*, 2006.
 - [94] J. Mats and J. Gunnarsson, "MathModelica - A new modeling and simulation environment for Mathematica," in *Proceedings fo the 3rd International Mathematica Symposium*, Linz, Austria, 1999.
 - [95] H. Pneumatics, "New simulation software available," vol. 63, ed: Penton Media, Inc., 2010, pp. 10-10.
 - [96] M. Najafi, R. Nikoukhah, S. Steer, and S. Furic, "New features and new challenges in modeling and simulation in scicos," in *Control Applications, 2005. CCA 2005. Proceedings of 2005 IEEE Conference on*, 2005, pp. 1109-1114.
 - [97] M. Najafi, S. Furic, and R. Nikoukhah, "Scicos: a general purpose modeling and simulation environment," in *Proceedings of the 4th International Modelica Conference*, Hamburg, 2005, pp. 367-374.
 - [98] P. Fritzson, "Introduction to Object-Oriented Modeling and Simulation with OpenModelica," 2006.

-
- [99] J. Jugo, "Real-Time control of a DC motor using Scilab and RTAI," Scilab INRIA Rocquencourt, Paris France 2004.
 - [100] S. L. Campbell, J. P. Chancelier, and R. Nikoukhah, "Implicit Scicos and Modelica," in *Modeling and Simulation in Scilab/Scicos*, ed: Springer New York, 2006, pp. 273-280.
 - [101] J. Zhao, Z. Li, J. Ding, L. Chen, Q. Wang, Q. Lu, *et al.*, "HIL Simulation of Aircraft Thrust Reverser Hydraulic System in Modelica," in *Proceedings of the 7th International Modelica Conference 2009*, 2009.
 - [102] R. Sinha, V.-C. Liang, C. J. J. Paredis, and P. K. Khosla, "Modeling and Simulation Methods for Design of Engineering Systems," *Computing and Information Science in Engineering*, vol. 1, pp. 84-91, 2001.
 - [103] J. Frenkel, C. Schubert, G. Kunze, and P. Fritzson, "Towards a Benchmark Suite for Modelica Compilers: Large Models," in *Proceedings 8th Modelica Conference*, Dersden, Germany, 2011.
 - [104] A. Ebner, A. Haumer, D. Simic, and F. Pirker, "Interacting Modelica using a Named Pipe for Hardware-in-the-loop Simulation," in *Proceedings of the 5th International Modelica Conference 2006*, 2006.
 - [105] P. Baracos, G. Murere, C. A. Rabbath, and W. Jin, "Enabling PC-Based HIL Simulation for Automotive Applications," in *Proceedings of the IEEE International Electric Machines and Drives Conference*, 2001.
 - [106] G. Ferretti, M. Gritti, G. Magnani, G. Rizzi, and P. Rocco, "Real-Time Simulation of Modelica Models under Linux/RTAI," in *Proceedings of the 4th International Modelica Conference 2005*, 2005.
 - [107] S. Abourida, C. Dufour, and J. Belanger, "Real-Time and Hardware-In-The-Loop Simulation of Electric Drives and Power Electronics: Process, problems and solutions," in *Proceeding of the 2005 International Power Electronics Conference*, 2005.
 - [108] S. Muller and H. Waller, "Efficient Integration Of Real-Time Hardware And Web Based Services Into Matlab," in *Proceedings of 11th European Simulation Symposium*, Erlangen, Germany, 1999.
 - [109] G. Ferretti, G. Magnani, P. Porriati, G. Rizzi, P. Rocco, and A. Rusconi, "Real-Time Simulation of a Space Robotic Arm," in *Proceedings of the 6th International Modelica Conference 2008*, 2008.
 - [110] M. Otter, C. Schlegel, and H. Elmqvist, "Modeling And Realtime Simulation Of An Automatic Gearbox Using Modelica," in *Proceedings of ESS'97 - European Simulation Symposium*, 1997.
 - [111] H. Elmqvist, S. E. Mattsson, H. Olsson, J. Andreasson, M. Otter, C. Schweiger, *et al.*, "Real-time Simulation of Detailed Automotive Models," in *Proceedings of the 3rd International Modelica Conference 2003*, 2003.
 - [112] D. Winkler and C. Guhmann, "Hardware-in-the-Loop Simulation of a hybrid electric vehicle using Modelica/Dymola," in *Proceedings of the 22nd International Battery, Hybrid and Fuel Cell Electric Vehicle Symposium and Exposition*, Yokohama, Japan, 2006, pp. 1054-1063.
 - [113] R. Bucher, S. Mannori, and T. Netter, "RTAI-Lab tutorial: Scilab, Comedi, and real-time control," 2008.
 - [114] T. Bellmann, "Interactive Simulations and advanced Visualization with Modelica," 2009.
 - [115] H. Elmqvist, M. Otter, D. Henriksson, B. Thiele, and S. E. Mattsson, "Modelica for Embedded Systems," in *Proceedings 7th Modelica Conference*, Como, Italy, 2009.
 - [116] M. Association. (2012, Nov 11). *Modelica Libraries-Modelica_DeviceDrivers*. Available: https://www.modelica.org/libraries/modelica_devicedrivers

-
- [117] Comedi. (2009, Nov 11). *Linux Control and Measurement Device Interface*. Available: www.comedi.org
 - [118] G. M. Amdahl, G. A. Blaauw, and F. P. Brooks, "Architecture of the IBM System/360," *IBM Journal of Research and Development*, vol. 8, pp. 87-101, 1964.
 - [119] D. Rubio. (2012, June 3). *Web application performance and scalability* [Book]. Available: <http://www.webforefront.com/performance/>
 - [120] S. J. Vaughan-Nichols, "Hard drive technology reaches a turning point," *Computer*, vol. 36, pp. 21-23, 2003.
 - [121] J. Barr, *Host Your Web Site In The Cloud: Amazon Web Services Made Easy*. United States of America: SitePoint, 2010.
 - [122] A. V. Mladen, "Cloud Computing - Issues, Research and Implementations," *Journal of Computing and Information Technology*, vol. 16, pp. 235-246, 2008.
 - [123] B. Peng, B. Cui, and X. Li, "Implementation on Issues of A cloud computing platform," *IEEE Computer Society Technical Committee on Data Engineering*, 2009.
 - [124] L. M. Surhone, M. T. Timplendon, and S. F. Marseken, *Wolfram Alpha*: VDM Verlag Dr. Mueller e.K., 2010.
 - [125] Microsoft. (2012, July 5). *Search 1,000s of templates*. Available: <http://office.microsoft.com/en-us/templates/>
 - [126] T. A. Davis, *MATLAB Primer, Eighth Edition*, 8, illustrated ed. United States: CRC Press, 2011.
 - [127] S. E. Mattsson, H. Elmqvist, and J. F. Broenink, "Modelica: An International Effort to Design the Next Generation Modelling Language," *Special issue on Computer Aided Control System Design, CACSD*, 1997.
 - [128] C. E. Wieman, W. K. Adams, and K. K. Perkins, "PhET: Simulations That Enhance Learning," *Science*, vol. 322, pp. 682-683, 2008.
 - [129] J. N. Harb, S. O. Durrant, and R. E. terry, "Use of the Kolb Learning Cycle and the 4MAT System in Engineering Education," *Engineering Education*, vol. 82, 1993.
 - [130] D. A. Kolb, *Experiential learning: experience as the source of learning and development*: Prentice-Hall, 1984.
 - [131] B. McCarthy, *The 4MAT system: teaching to learning styles with right/left mode techniques*: EXCEL, 1981.
 - [132] Q. Shen, Y. Jiang, and X. Su, "Exploration on Practice Teaching in Computer Major," *International Conference on Educational and Information Technology*, 2010.
 - [133] X. Chen, "Innovative Experiment Teaching in Engineering Practice Courses," *2nd International Conference on Education Technology and Training*, 2009.
 - [134] E. Winsberg, "Simulated Experiments: Methodology for a Virtual World," *Philosophy of Science*, vol. 70, pp. 105-125, 2003.
 - [135] N. Finkelstein, W. Adams, C. Keller, K. Perkins, and C. Wieman, "High-Tech Tools for Teaching Physics: the Physics Education Technology Project," *Journal of Online Learning and Teaching*, vol. 2, 2006.
 - [136] A. Jimoyiannis and V. Komis, "Computer simulations in physics teaching and learning: a case study on students' understanding of trajectory motion," *Computers & Education*, vol. 36, pp. 183-204, 2001.
 - [137] Instructables. (2012). *Instructables*. Available: <http://www.instructables.com>
 - [138] Z. Yanshan, Y. F. Fah, J. Singh, L. Ting, and G. Lei, "A knowledge-based web platform for collaborative physical system modeling and simulation," *Computer Applications in Engineering Education*, pp. n/a-n/a, 2013.
 - [139] Drupal. (2012). *Drupal Community Documentation*. Available: <http://drupal.org/documentation>

-
- [140] C. Heng, "3 Ways of Putting a Search Engine on Your Website," 2012.
 - [141] M. S. Keller, "Take Command: cron: Job Scheduler," *Linux J.*, vol. 1999, p. 15, 1999.
 - [142] Drupal, "Module developer's guide," *Drupal Community Documentation*, 2012.
 - [143] M. volkel, M. Krotzsch, D. Vrandecic, H. Haller, and R. Studer, "Semantic Wikipedia," in *Proceedings of the 15th international conference on World Wide Web*, Edinburgh Scotland, 2006.
 - [144] Flickr. (2012). *Flickr*. Available: www.flickr.com
 - [145] J. Fons, V. Pelechano, M. Albert, and Ó. Pastor, "Development of Web Applications from Web Enhanced Conceptual Schemas
- Conceptual Modeling - ER 2003." vol. 2813, I.-Y. Song, S. Liddle, T.-W. Ling, and P. Scheuermann, Eds., ed: Springer Berlin / Heidelberg, 2003, pp. 232-245.
- [146] G. GWT, "Large scale application development and MVP," *Google Developers*, 2010.
 - [147] Modelica and M. Association, "Modelica Library 3.1," *Modelica and Modelica Association*, 1996.
 - [148] gwt-g2d. gwt-g2d [Online]. Available: <http://code.google.com/p/gwt-g2d/>
 - [149] Modelica, "Modelica Specification version 3.1," *Modelica and Modelica Association*, 2010.
 - [150] P. Fritzson, "OpenModelica User Guide," 2011.
 - [151] H. Eriksson, H. Magnusson, P. Fritzson, and A. Pop, "3D Animation and Programmable 2D Graphics for Visualization of Simulations in OpenModelica," in *Proceedings of the 49th Scandinavian Conference on Simulation and Modeling (SIMS' 2008)*, Norway, 2008.
 - [152] H. Eriksson, "Advanced OpenModelica plotting package for Modelica," Master Thesis Master, Department of Computer and Information Science Linköping University, Linköping University, 2008.
 - [153] V. Engelson, H. Larsson, and P. Fritzson, "A Design, Simulation and Visualization Environment for Object-Oriented Mechanical and Multi-Domain Models in Modelica," in *Proceedings of 1999 IEEE International Conference on Information Visualization*, London, 1999, pp. 188-193.
 - [154] P. Fritzson, V. Engelson, A. Idebrant, P. Aronsson, H. Lundvall, P. Bunus, *et al.*, "Modelica—A Strongly Typed System Specification Language for Safe Engineering Practices," in *Proceedings of the SimSAFE Conference*, Kralskoga, Sweden, 2004.
 - [155] L. Saldamli, P. Fritzson, and B. Bachmann, "Extending Modelica for Partial Differential Equations," in *the 2nd International Modelica Conference*, Oberpfaffenhofen, Germany, 2002.
 - [156] L. Saldamli and P. Fritzson, "Object-oriented Modeling with Partial Differential Equations," in *Proceedings of Modelica Workshop 2000*, 2000.
 - [157] Z. Li, L. Zheng, and H. Zhang, "Solving PDE Models in Modelica," presented at the 2008 International Symposium on Information Science and Engineering, 2008.
 - [158] N. Nurseitov, M. Paulson, R. Reynolds, and C. Izurieta, "Comparison of JSON and XML Data Interchange Formats: A Case Study," 2009.
 - [159] B. Sun, "Comparison and Research of JSON and XML," *Inner Mongolia Science Technology & Economy*, vol. 12, pp. 122-126, 2009.
 - [160] JSON. (2012). *JSON Examples*. Available: www.json.org
 - [161] Y. Zhu, H. Hu, G. Xu, and Z. Zhao, "Hardware-in-the-Loop Simulation of Pure Electric Vehicle Control System," in *Proceedings of 2009 International Asia Conference on Informatics in Control, Automation and Robotics*, 2009.
 - [162] H. Elmqvist, S. E. Mattsson, and H. Olsson, "New Methods for Hardware-in-the-loop Simulation of Stiff Models," in *Proceedings of the 2nd International Modelica Conference 2002*, 2002.

-
- [163] ADI, "Distributed HIL Simulation," *Applied Dynamics International*, 2005.

Appendix A

Modelica Lexical Definition

```
options{
  STATIC = false;
  MULTI = true;
  NODE_DEFAULT_VOID = true;
  NODE_PREFIX = "OM";
  VISITOR = true;
}
PARSER_BEGIN(ModelicaParser)
package com.infosci.proteus.modelica.parser;
import java.util.List;
import java.util.LinkedList;
import java.util.Arrays;
public class ModelicaParser{
  public static void main(String[]args)throws ParseException, TokenMgrError{
    ModelicaParser parser = new ModelicaParser(System.in);
    SimpleNode root = parser.stored_definition();
    root.dump("");
  }
}
PARSER_END(ModelicaParser)
SKIP:{
  " "
  | "\t"
  | "\n"
  | "\r"
  | <"/"/(<~["\n", "\r"]>)*(<"\n"
  | "\r"
  | "\r\n">
  | <"/*"(<~["*"]>)*"("(<~["*", "/">
  | ~["*", "/">
}
TOKEN:{
  <ALGORITHM:"algorithm">
  | <AND:"and">
  | <ANNOTATION:"annotation">
  /* | <ASSERT: "assert"> assert can also appear as a function, making it a
keyword breaks it */
  | <BLOCK:"block">
  | <BREAK:"break">
  | <CLASS:"class">
  | <CONNECT:"connect">
  | <CONNECTOR:"connector">
  | <CONSTANT:"constant">
  | <CONSTRAINEDBY:"constrainedby">
  /* | <DER: "der"> der can also appear as a function, making it a keyword
breaks it */
```

```

| <DISCRETE:"discrete">
|   <EACH:"each">
|   <ELSE:"else">
|   <ELSEIF:"elseif">
|   <ELSEWHEN:"elsewhen">
|   <ENCAPSULATED:"encapsulated">
|   <END:"end">
|   <ENUMERATION:"enumeration">
|   <EQUATION:"equation">
|   <EXPANDABLE:"expandable">
|   <EXTENDS:"extends">
|   <EXTERNAL:"external">
|   <FALSE:"false">
|   <FINAL:"final">
|   <FLOW:"flow">
|   <FOR:"for">
|   <FUNCTION:"function">
|   <IF:"if">
|   <IMPORT:"import">
|   <IN:"in">
|   <INITIAL:"initial">
|   /* initial can also appear as a function, making it a keyword breaks it */
| <INNER:"inner">
|   <INPUT:"input">
|   <LOOP:"loop">
|   <MODEL:"model">
|   <NOT:"not">
|   <OR:"or">
|   <OUTER:"outer">
|   <OUTPUT:"output">
|   <PACKAGE:"package">
|   <PARAMETER:"parameter">
|   <PARTIAL:"partial">
|   <PROTECTED:"protected">
|   <PUBLIC:"public">
|   <RECORD:"record">
|   <REDECLARE:"redeclare">
|   <REPLACEABLE:"replaceable">
|   <RETURN:"return">
|   <STREAM:"stream">
|   <THEN:"then">
|   <TRUE:"true">
|   <TYPE:"type">
|   <WHEN:"when">
|   <WHILE:"while">
|   <WITHIN:"within">
}
TOKEN:{
  <REL_OP:"<"
  | "<="
  | ">"
  | ">="
  | "=="
  | "<>"
  | <PLUS:"+">

```

```

| <MINUS: "-">
| <DOT_PLUS: ".+">
| <DOT_MINUS: "-.">
| <ADD_OP: "+">
| <SUB_OP: "-">
| <MULT_OP: "*">
| <DIVIDE_OP: "/">
| <DOT_DIVIDE_OP: "./">
| <MUL_OP: <TIMES>>
| <DIVIDE>
| <DOT_TIMES>
| <DOT_DIVIDE>>
}
TOKEN: {
  <IDENT: <NONDIGIT> (<NONDIGIT> | <DIGIT>)* | <Q_IDENT>>
  <#Q_IDENT: "\"" (<Q_CHAR>
  <S_ESCAPE>)+ "\"">
  <#NONDIGIT: ["_", "a"-"z", "A"-"Z"]>
  <STRING: "\"" (<S_CHAR>
  <S_ESCAPE>)* "\"">
  <#S_CHAR: ~["\"", "\\"]>
  <#Q_CHAR: ~["\"", "\\"]>
  <#S_ESCAPE: "\\\">
  "\\\">
  "\\?">
  "\\\">
  "\\a">
  "\\b">
  "\\f">
  "\\n">
  "\\r">
  "\\t">
  "\\v">
  <#DIGIT: ["0"-"9"]>
  <UNSIGNED_INTEGER: (<DIGIT>)+>
  <UNSIGNED_NUMBER: (<UNSIGNED_INTEGER> ("." (<UNSIGNED_INTEGER>)?)? ("e"
  "E") ("+"
  "-")? <UNSIGNED_INTEGER>)?>
  "." <UNSIGNED_INTEGER> ("e"
  "E") ("+"
  "-")? <UNSIGNED_INTEGER>)?>
  //1 Nov, 2010 Change the Unsignd_number representation to support double number
  like ".454454"
  // |<BOOL: "true"|"false">
  // |<INTEGER: <UNSIGNED_INTEGER>>
  // |<REAL: (<UNSIGNED_INTEGER>)? "." (<UNSIGNED_INTEGER>)? ("e" | "E") ("+" |
  "-")? <UNSIGNED_INTEGER>)?>
}

```

Appendix B

Modelica Grammar Definition

```
/*
*****
*
*           Stored Definition - Within
*
*****
**/
OMStoredDefinition stored_definition()#StoredDefinition:{
  boolean final_ = false;
  String s;
  OMClassDefinition clsdef;
}
{
  ((<WITHIN>{
    jjtThis.within = true;
  }
  (s = name(){
    jjtThis.name = s;
  }
  )?";")?(<FINAL>{
    final_ = true;
  }
  )?clsdef = class_definition(){
    clsdef.final_ = final_;
    jjtThis.classDefinitions.add(clsdef);
  }
  ";"*){
    return jjtThis;
  }
}
/*
*****
*
*           Class Definition
*
*****
**/
OMClassDefinition class_definition()#ClassDefinition:{
  Token opt = null, t;
}
{
  (<ENCAPSULATED>{
    jjtThis.encapsulated = true;
  }
  )?(<PARTIAL>{
    jjtThis.partial = true;
  }
  )?(t = <CLASS>
  | t = <MODEL>
  | t = <RECORD>
  | t = <BLOCK>
  | (opt = <EXPANDABLE>)?t = <CONNECTOR>
```

```

| t = <TYPE>
| t = <PACKAGE>
| t = <FUNCTION>){
    jjtThis.restriction = (opt != null?opt.image:"")+t.image;
}
jjtThis.classSpecifier = class_specifier(){
    return jjtThis;
}
}
/*
*****
*                                     Class Definition
*****
**/
OMClassSpecifier class_specifier()#ClassSpecifier:{
    Token t;
}
{
    (LOOKAHEAD(2)t = <IDENT>{
        jjtThis.name = t.image;
    }
    jjtThis.description = string_comment()jjtThis.composition =
composition()<END><IDENT>{
    jjtThis.endTag = true;
    jjtThis.selection = 0;
}
| LOOKAHEAD(3)t = <IDENT>{
    jjtThis.name = t.image;
    jjtThis.selection = 1;
}
    ="jjtThis.typePrefix
=base_prefix()jjtThis.refName=name()(jjtThis.arraySubs=array_subscripts())?(jj
tThis.classModification = class_modification())?jjtThis.comment = comment()
    | LOOKAHEAD(3)t = <IDENT>{
        jjtThis.selection = 2;
        jjtThis.name = t.image;
    }
    ="<ENUMERATION>"("((jjtThis.omEnumlist=enum_list()
    | ":"{jjtThis.allTag = true;}))?"")jjtThis.comment =comment()
    | t = <IDENT>{
        jjtThis.selection = 3;
        jjtThis.name = t.image;
    }
    ="der"("jjtThis.refName = name()", "<IDENT>(", "<IDENT>)*")jjtThis.comment
= comment()
    | <EXTENDS>{
        jjtThis.selection = 4;
        jjtThis.extends_ = true;
    }
    t = <IDENT>{
        jjtThis.name = t.image;
    }
    (jjtThis.classModification = class_modification())?jjtThis.description =
string_comment()jjtThis.composition = composition()<END><IDENT>{
    jjtThis.endTag = true;

```

```

    }
    ){
        return jjtThis;
    }
}
OMTypePrefix base_prefix():{
    OMTypePrefix t;
}
{
    t = type_prefix(){
        return t;
    }
}
OMEnumList enum_list()#EnumList:{
    OMEnumLiteral el;
}
{
    (el = enumeration_literal(){
        jjtThis.enumList.add(el);
    }
    ("el = enumeration_literal(){
        jjtThis.enumList.add(el);
    }
    )*){
        return jjtThis;
    }
}
OMEnumLiteral enumeration_literal()#EnumLiteral:{
    Token t;
}
{
    (t = <IDENT>{
        jjtThis.name = t.image;
    }
    jjtThis.comment = comment()){
        return jjtThis;
    }
}
}
OMComposition composition()#Composition:{
    Token t;
    OMElementList el;
    OMEquationSection es;
    OMAgorithmSection as;
}
{
    (el = element_list(){
        jjtThis.elementList = el;
    }
    (t = <PUBLIC>el = element_list(){
        jjtThis.publicElementLists.add(el);
    }
    | t=<PROTECTED>el = element_list(){
        jjtThis.protectedElementLists.add(el);
    }
    | LOOKAHEAD(2)es = equation_section(){

```

```

        jjtThis.equationSections.add(es);
    }
    | as = algorithm_section(){
        jjtThis.algorithmSections.add(as);
    }
    )*(<EXTERNAL>(language_specification())?(external_function_call())?(annotation())?";")?(jjtThis.annotation = annotation();""){
        return jjtThis;
    }
}
String language_specification():{
    Token t;
}
{
    t = <STRING>{
        return t.image.substring(1, t.image.length()-1);
    }
}
OMExternalFuncCall external_function_call()#ExternalFuncCall:{
    Token t;
}
{
    ((LOOKAHEAD(2)jjtThis.componentRef = component_reference("=")?t = <IDENT>{
        jjtThis.funcName = t.image;
    }
    "(" (jjtThis.expList = expression_list())?"")){
        return jjtThis;
    }
}
}
OMElementList element_list()#ElementList:{
    OMElement e;
    OMAnnotation a;
}
{
    (e = element(){
        jjtThis.elements.add(e);
    }
    ";"
    | a = annotation(){
        jjtThis.annotations.add(a);
    }
    ";" )*{
        return jjtThis;
    }
}
}
OMElement element()#Element:{{}
    (jjtThis.importClause = import_clause()
    | jjtThis.extendsClause = extends_clause()
    | (<REDECLARE>{
        jjtThis.redeclare = true;
    }
    )?(<FINAL>{
        jjtThis.final_ = true;
    }
    )?(<INNER>{

```

```

        jjtThis.typing= "inner";
    }
    )?(<OUTER>{
        jjtThis.typing = "outer";
    }
    )?((jjtThis.classDefinition = class_definition()
    | jjtThis.componentClause = component_clause())
    | <REPLACEABLE>{
        jjtThis.replaceable = true;
    }
    (jjtThis.classDefinition = class_definition()
    | jjtThis.componentClause = component_clause()))(jjtThis.constrainingClause =
    constraining_clause()jjtThis.comment = comment())?){
        return jjtThis;
    }
}
OMImportClause import_clause()#ImportClause:{
    Token t;
    String alias;
}
{
    (<IMPORT>(LOOKAHEAD(2)t = <IDENT>"="jjtThis.name = name(){
        jjtThis.alias = t.image;
    }
    | jjtThis.name = name()(<DOT_TIMES>{
        jjtThis.includeSub = true;
    }
    )?)jjtThis.comment = comment()){
        return jjtThis;
    }
}
/*
*****
*
*                               Extends
*****
*
*/
OMExtendsClause extends_clause()#ExtendsClause:{{
    (<EXTENDS>jjtThis.name = name()(jjtThis.classModification =
    class_modification())?(jjtThis.annotation = annotation())?){
        return jjtThis;
    }
}
}
OMConstrainingClause constraining_clause()#ConstrainingClause:{{
    (<CONSTRAINEDBY>jjtThis.name = name()(jjtThis.classModification =
    class_modification())?){
        return jjtThis;
    }
}
}
/*
*****
*
*                               Component Clause

```

```

*****
*
*/
OMComponentClause component_clause()#ComponentClause:[]{
    jjtThis.typePrefix = type_prefix()jjtThis.typeName =
    type_specifier()(jjtThis.arraySubscripts =
    array_subscripts())?jjtThis.componentList = component_list(){
        return jjtThis;
    }
}
OMTypePrefix type_prefix()#TypePrefix:[]{
    //11 Nov, 2010. Stream is new in Modelica library 3.1.
    (<FLOW>{
        jjtThis.flow_stream = "flow";
    }
    | <STREAM>{
        jjtThis.flow_stream = "stream";
    }
    )?(<DISCRETE>{
        jjtThis.variability = "discrete";
    }
    | <PARAMETER>{
        jjtThis.variability = "parameter";
    }
    | <CONSTANT>{
        jjtThis.variability = "constant";
    }
    )?(<INPUT>{
        jjtThis.causality = "input";
    }
    | <OUTPUT>{
        jjtThis.causality = "output";
    }
    )?{
        return jjtThis;
    }
}
String type_specifier():{
    String n;
}
{
    n = name(){
        return n;
    }
}
OMComponentList component_list()#ComponentList:{
    OMComponentDecl cd;
}
{
    (cd = component_declaration(){
        jjtThis.componentDecls.add(cd);
    }
    | ", "cd = component_declaration(){
        jjtThis.componentDecls.add(cd);
    }
    )
}

```

```

    *){
        return jjtThis;
    }
}
OMComponentDecl component_declaration()#ComponentDecl:{{
    (jjtThis.decl = declaration()(jjtThis.condAttr =
conditional_attribute())?jjtThis.comment = comment()){
        return jjtThis;
    }
}
OMCondAttr conditional_attribute()#CondAttr:{{
    (<IF>jjtThis.expression = expression()){
        return jjtThis;
    }
}
OMDeclaration declaration()#Declaration:{
    Token t;
}
{
    (t = <IDENT>{
        jjtThis.name = t.image;
    }
    (jjtThis.arraySubscripts = array_subscripts())?(jjtThis.modification =
modification())?){
        return jjtThis;
    }
}
}

```

Appendix C

Basic Graphic Elements in Modelica

Line Annotation

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

Polygon Annotation

```
record Polygon
  extends GraphicItem;
  extends FilledShape;
  Point points[:];
  Smooth smooth = Smooth.None    "Spline outline";
end Polygon;
```

Rectangle Annotation

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

Ellipse Annotation

```
record Ellipse
  extends GraphicItem;
  extends FilledShape;
  Extent extent;
  Real startAngle(quantity="angle", unit="deg")=0;
  Real endAngle(quantity="angle", unit="deg")=360;
end Ellipse;
```

Text Annotation

```
record Text
  extends GraphicItem;
  extends FilledShape;
  Extent extent;
  String textString;
  Real fontSize = 0 "unit pt";
  String fontName;
  TextStyle textStyle[:];
  TextAlignment horizontalAlignment = TextAlignment.Center;
end Text;
```

Bitmap Annotation

```
record Bitmap
  extends GraphicItem;
  Extent extent;
  String fileName "Name of bitmap file";
  String imageSource "Base64 representation of bitmap";
end Bitmap;
```