AN XML-BASED DIAGRAMMATIC DYNAMIC MODELING AND SIMULATION SYSTEM

By

HYUNJU SHIM

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2003

To my parents,

and my two brothers

ACKNOWLEDGMENTS

I would like to show my deepest gratitude to my advisor, Dr. Paul A. Fishwick. He inspired me with insightful comments and suggestions in this interesting research and provided invaluable guidance during my studies. I also would like to show my thanks to Dr. Joachim Hammer and Dr. George Essl for serving as my committee members.

I really appreciate my research colleagues, Minno Park and Jinho Lee, for their companionship. They always helped me and offered valuable discussions on my research. They also provided a warm family-like environment in our lab. I also would like to thank Taewoo Kim for his kind assistance and guidance when I first got oriented in our lab.

Above all, I would like to show my deepest thanks and respect to my wonderful parents and two brothers. They have always loved me as I am and trusted me in any situation. Without their encouragement, it would have been impossible to carry on my studies and research for my Master of Science degree.

TABLE OF CONTENTS

page

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

AN XML-BASED DIAGRAMMATIC DYNAMIC
MODELING AND SIMULATION SYSTEM

By

Hyunju Shim

August, 2003

Chair: Paul A. Fishwick
Major Department: Computer and Information Science and Engineering

The emergence of eXtensible Markup Language (XML) brought a new way of representing information and knowledge on the World Wide Web. The XML was also found to be very useful for modeling and configuring graphical elements and opened a new way to share them over the Web. Much research and XML-based technology is used for modeling and visualization in Extensible 3D (X3D) and Scalable Vector Graphic (SVG). The primary goal of this thesis was to develop an effective way to represent a model in XML, especially SVG. A secondary goal was to construct a powerful modeling and simulation framework using XML–based technology.

We constructed a *rube*2D modeling and simulation framework to broaden the paradigm and methodology of *rube* into the 2D models, especially for the diagrammatic dynamic models. The precursor of *rube*2D, *rube*, was developed as a modeling and simulation framework for 3D models.

The models in *rube*2D can be viewed in two different perspectives. The model can be understood by its dynamic behavior over time. It also can be described according to its physical appearance or metaphoric presentation. The two perspectives of the model are defined in two separate parts of *rube*2D. The model topology is defined in *Multimodel eXchange Language* (MXL). The model presentation is defined using SVG. Separating the model topology from the presentation makes the model topology independent from presentation and vice versa. This independence provides great advantages such as effectiveness, flexibility, and aesthetic computing in the modeling and simulation and this is what distinguishes *rube*2D from similar works.

For creating the simulation code from the model topology, Model Translation Engines is defined in XSLT stylesheet. To represent heterogeneous model types in MXL into homogeneous block model, *Dynamics eXchange Language* (DXL) is defined. DXL is an assembly-like language between a high level model description language, MXL and the executable simulation code. Model Translation Engines translate MXL into DXL and DXL into a simulation JavaScript code. The system being modeled in MXL can be simulated using *SimPackJ/S*, which provides simulation code libraries.

Model Fusion Engine, which is written in the eXtensible Stylesheet Language Transformations (XSLT), is developed to merge the model presentation defined in SVG with the simulation code generated from the model topology, MXL. Finally a dynamic SVG that behaves as defined in model topology is generated as a result of model fusion.

The final output of *rube*2D is an SVG that has same geometry as the original SVG, and dynamic behavior as defined in the model topology file, MXL.

CHAPTER 1
INTRODUCTION

1.1 Motivation

The primary motivation for my research was the the possibility of extending the power of modeling and simulation using *rube* in 2D modeling. The modeling and simulation framework *rube* is based on eXtensible Markup Language (XML) technologies and has a distinct modeling and simulation methodology of separating model topology from presentation.

The research on 2D modeling and simulation using *rube*2D mainly focuses on two issues. One issue is how to present a model in a 2D world while keeping the XML-based architecture. The answer came from the existing XML for 2D graphics. By having Scalarable Vector Graphics (SVG) with fully defined specification, SVG model presentation in *rube*2D became a very solid process.

The process of obtaining a simulation code from a model topology is well defined and developed in *rube*. The second issue of 2D modeling and simulation using *rube*2D is how to make extensive use of the simulation code for simulating a model defined in SVG. The answer came from one of the XML technologies, eXtensible Stylesheet Language Transformations (XSLT). By defining transforming rules and merging rules between the original SVG and simulation code in the XSLT stylesheet, dynamic SVG with JavaScript simulation code associated can be obtained. Simulation of the dynamic SVG can be easily achieved by simply running the final SVG.

As an extended interest, MXL creation tool with Graphic User Interface (GUI) is implemented to facilitate modelers. This tool provides a way to create MXL from a presentation, SVG.

1.2 Main Contribution

*rube*2D is an open-source framework. Modelers can do modeling and simulation using *rube*2D for free. Also, modelers can copy and modify a whole or part of the source codes in any way. *rube*2D is XML-based. Powerful XML technologies, such as XSLT stylesheet, the XSLT processor, and Web-based modeling are used in *rube*2D.

The methodology used in *rube*2D, separating model semantics from presentation, provides excellent features to the modeling and simulation in 2D. First, an aesthetic computing and modeling in 2D can be achieved easily by the existence of an independent model presentation from its semantics. Second, several different metaphors for the model presentation can be applied over the same model topology. Finally, integration of the scene with a dynamic model is much easier in the *rube*2D environment.

1.3 Organization of the Thesis

Chapter 2 discusses background knowledge of *rube*2D. Chapter 3 introduces the related modeling and simulation works. A detailed discussion of *rube*2D is presented in Chapter 4. The paradigm and methodology of *rube*2D is introduced followed by an explanation of the development environment and overall framework. Also, a detailed discussion about the contribution of *rube*2D is addressed in Chapter 4. Chapter 5 introduces modeling in *rube*2D divided into two parts: model topology design and model presentation design. The model simulation process with the Model Fusion Engine is introduced in Chapter 6. Conclusions are given in Chapter 7.

CHAPTER 2
BACKGROUND KNOWLEDGE

Basic ideas and knowledge underlying *rube*2D, which is an XML-based diagrammatic dynamic modeling and simulation framework, are introduced in this chapter. The eXtensible Markup Language (XML) is reviewed first followed by Scalar Vector Graphics (SVG) which is 2D graphical XML. Next, XML technology eXtensible Stylesheet Language Transformation (XSLT), which plays an important role in *rube* and *rube* 2D architectures is discussed. Finally, *rube,* the precursor of *rube* 2D, is explained in the last section.

2.1 Extensible Markup Language (XML)

The definition of XML from the abstract of the XML 1.0 specification [1] is:

> The Extensible Markup Language (XML) is a subset of SGML that is completely described in this document. Its goal is to enable generic SGML to be served, received, and processed on the Web in the way that is now possible with HTML. XML has been designed for ease of implementation and for interoperability with both SGML and HTML.

As described in the XML specification, XML is directly derived from Standard Generalized Markup language (SGML). Although SGML is very powerful and useful, particularly for dealing with large quantity of structured data, it is complex and expensive. The complexity of SGML makes it inconvenient for use over the Internet. The main design goal of XML is to extend the power of SGML to the Web in a way that is now possible with HTML. The XML simplifies SGML for ease of implementation while it inherits the power of SGML.

The XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere. While HTML is mainly about how to represent information, XML mainly deals with the nature of the information. This feature of XML lets the programmer manipulate the data in an XML document for the data representation and also for other purposes. The XML also provides a way to define one's own tags and structure of documents. The XML schema provides a means for defining the structure, contents, and semantics of documents.

Unlike HTML, XML separates information from details by how it is presented. This enables information to be rendered or used appropriately for a variety of devices. This also gives great flexibility for design and redesign of XML documents.

Another great power of XML comes from its transformability. Through the XSLT process, one XML document can easily converted into another XML document. Using XSLT, one domain of XML can be translated into another domain of XML without a complicated process.

The XML features previously described give strong power to XML and motivate many research groups and enterprises to use XML as a tool or a software environment. The last two features or advantages of XML, separation of data from presentation and transformability, are the main features of *rube* architecture.

2.2 Extensible Stylesheet language Transformation (XSLT)

The XSLT definition in the specification [2] gives a clear idea what it is:

> XSLT is designed for use as part of XSL, which is a stylesheet language for XML. In addition to XSLT, XSL includes an XML vocabulary for specifying formatting. XSL specifies the styling of an XML document by using XSLT to describe how the document is transformed into another XML document that uses the formatting vocabulary.

XSLT 1.0 published on 16 November 1999. The XSLT describes how an XML document or several XML documents are translated into another XML document. The XSLT plays a very important role for data transformation between two different XML applications.

Encoding interchanging data between different organizations or different applications in XML does not mean automatic data conversion between two different domains. Encoding in XML means it is easy to create and read the data and it is easy to write programs which process XML documents. Some kind of intermediate process is needed to convert one XML document into another form of an XML document. The XSLT stylesheet translates an XML document into another XML document based on template rules which describe how each element must be processed for transformation. A transformation expressed in XSLT is called a stylesheet.

For an XML document transformation, the XSLT processor reads both the XSLT stylesheet and XML documents. The XSLT processor outputs new XML documents by reading the XML document and applying rules described in the XSLT stylesheet. The well-known XSLT processors are SAXON [3] developed by Michael Kay and XALAN [4] developed by the Apache Software Foundation.

In *rube* architecture, two kinds of XML, MXL and DXL, are defined for model types and dynamic behavior of a model description, and the XSLT stylesheet is defined for translation needed for the modeling and simulation process, which will be discussed in Chapter 2 and Chapter 4 in detail.

2.3 Scalable Vector Graphic (SVG)

In the Scalable Vector Graphic (SVG) 1.0 specification, SVG is defined as a sublanguage of XML describing a two-dimensional vector or vector/raster mixed

graphics [5]. Three types of graphic objects in SVG are vector graphic shapes, image, and text. The SVG is a W3C recommendation released 4 September 2001. By being written in XML, a SVG builds on strong foundation and gains many advantages from XML, such as standardization, internationalization, powerful structuring capability, an object model, and so forth.

The SVG drawings are interactive and dynamic. Interactive means that SVG drawings can interact with a user via input devices such as a mouse. Dynamic means the SVG drawing can be changed via a script or declarative 'animation' element. Sophisticated interactive and dynamic SVG applications are made possible by use of Web-based script languages, such as JavaScript, which accesses SVG DOM (Document Object Model) and with a rich set of event handlers. The SVG DOM provides complete access to all elements and attributes of the SVG document. Thus all elements and attributes of the SVG document are easily accessible and dynamically changeable after it is loaded into a web browser. Also, SVG works well across platforms and output resolutions.

With all the strong advantages previously described and the openness of SVG, SVG has emerged as a great interest among Web designers and computer graphic communities. *rube* has extended its visual presentation level not only in 3D but also into 2D with SVG support. SVG can fit into a model presentation component of *rube* architecture without any difficulties since SVG itself is a sublanguage of XML and *rube* is XML-based architecture.

2.4 *rube*

In this section, we will discuss software modeling and a visualization framework called *rube* developed by *rube* research and development team in the Computer and

Information Science and Engineering Department at the University of Florida. *rube* is a 3D XML-based customized modeling and simulation framework for dynamic models. *rube* framework encompasses model simulation, as well as modeling of a dynamic model. In Section 2.4.1, the paradigm and methodology of *rube* is introduced. *rube* framework is described in Section 2.4.2. Finally, the modeling and simulation process in *rube* is discussed in Section 2.4.3.

2.4.1 Paradigm and Methodology

The main goal of *rube* is creating a modeling and simulation methodology that supports a separation of a dynamic model semantic from presentation and visualization [6]. *rube* methodology creates great potential into system modeling and simulation. A consideration of metaphor will play a great role in *rube*. *rube* paradigm allows the model developer total freedom in the choice of metaphor [7]. *rube* also prompts the integration of an aesthetic aspect of a model into modeling and simulation.

rube evolved from its precursor *Object-Oriented Physical Multimodeling* (OOPM). With OOPM, programs are multimodels defined as a hierarchically connected set of dynamic models. Each model type in a multimodel is a basic dynamic behavioral model type, such as Finite State Machine (FSM), Functional Block Model (FBM), System Dynamics Model (SDM), Equation Constraint Model (ECM), Petri Net (PNET), and others [8]. As a successor of OOPM, *rube* encompasses single models, as well as multimodels. The single dynamic behavioral models are freely and easily combined into or glued together into multimodeling in *rube*.

2.4.2 Framework

The overall structure of XML-based *rube* architecture is shown in Figure 2-1. *rube* is mainly a 3D XML-based framework for dynamic models. The model in *rube* is

described in two separate files: a scene file which describes model presentation in X3D, and a model file which defines the semantic of model in MXL. X3D [9] is XML version of Virtual Reality Modeling Language (VRML). X3D is able to express geometry and behavior capabilities of VRML 97. MXL stands for Multimodel eXchange Language, which defines the topology and dynamic behavior of the model type. MXL will be discussed in Chapter 5 in detail.

In *rube* architecture, the model file, which is written in MXL, is translated into assembly level model language DXL by an XSLT stylesheet. Overall structure of *rube* framework is shown in the Figure 2-1.

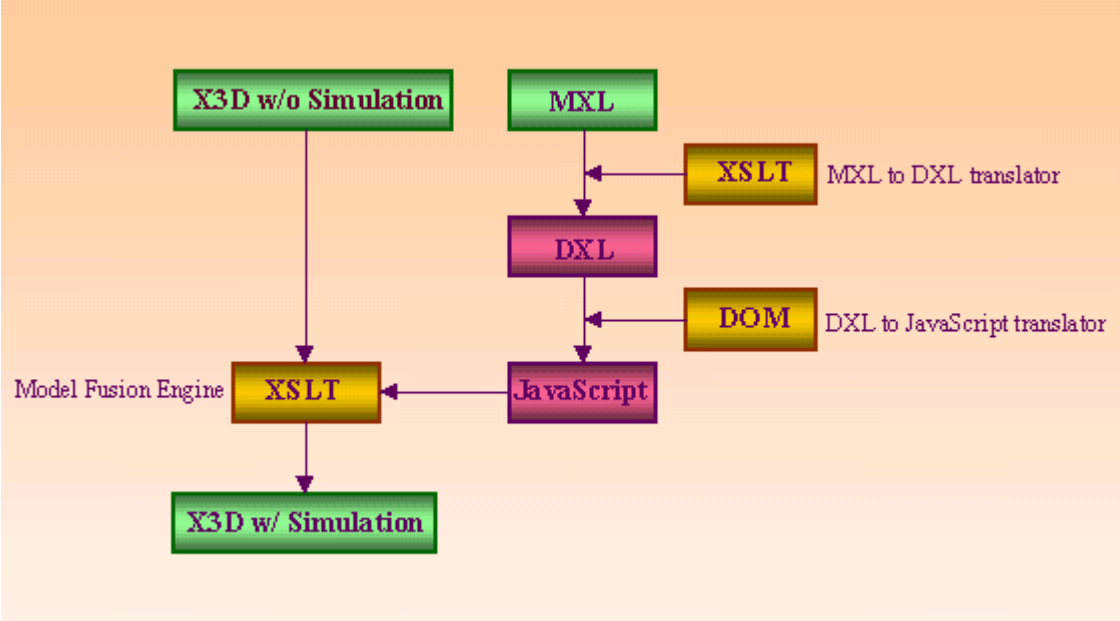

Figure 2-1. The XML-based *rube* framework structure

DXL is another model language which stands for Dynamic eXchange Language. DXL represents a multimodel containing more than one heterogeneous sub-model type within homogeneous block diagrams. DXL acts like assembly level languages between MXL and a simulation code referencing SimPackJ/S library [10]. SimPack J/S is a collection of JavaScript libraries for computer simulation derived from SimPack. DXL is translated

into a simulation code using Document Object Model (DOM). Currently the simulation code generated from DXL is written in JavaScript. Generating a Java version of the simulation code is now under development. The simulation code can be a stand-alone processor, which generates simulation output of the model semantic in MXL.

2.4.3 Modeling and Simulation in *rube*

In the XML-based *rube* framework, the model presentation and model semantic are described with XML. However, to visualize the model in any way, the X3D model presentation file needs to have a mechanism to visualize the X3D contents describing the model geometry since no X3D viewer is available. Fortunately, with no surprise, there many kinds of translators exist which transform X3D into VRML and vice versa. Thus developers can create the model presentation file in VRML and translate it into X3D. X3D with simulation code can be transformed into VRML for visualization.

By using the simulation output generated from the simulation code written in JavaScript, X3D or VRML model presentation can be accessed dynamically and simulated as the user sets the relationship between the simulation output and model geometry.

CHAPTER 3
RELATED WORK

This chapter discusses related research work in modeling and simulation and the model and software visualization. Among various modeling and simulation related work, XML technologies [1-5] based works are discussed in this chapter.

SVG and X3D are XML graphic languages and both provide rich elements of graphical components. Many research groups are adapting SVG or X3D for modeling and visualization. Modeling and visualization in SVG and X3D are discussed first and Visual Simulation Environment (VSE) is introduced next. In the last section, the open source SVG creation tool, Sodipodi, which facilitates users with great functionality and easiness, is introduced.

3.1 Scalable Vector Graphic (SVG) Visualization and Modeling

Visualization using SVG is proposed by Domokos and Varro[11]. The main features of this visualization framework include:

- SVG-based batch visualization framework for modeling languages defined by metamodeling techniques
- Combining XML standards with existing graph transformation and graph drawing technologies
- An open, tool-independent architecture

Their overall aim is to visualize a concrete model such as Petri Net by transforming it into an SVG representation, which can be rendered by Web browsers. The visualizing process involves transforming from an abstract representation of model to a concrete SVG-based textual format model. The motamodel first transformed into XML Metadata

Interchange(XMI) by their "layout" translator. This transformed XML from XMI is translated into SVG for the model visualization. XSLT stylesheet is applied over XMI and generate SVG.

The *rube* further extends use of the technology of XML in modeling and simulation. SVG in *rube* 2D not only visualizes a system model but also transforms into a dynamic SVG, which is accessible to a simulation code of a dynamic behavior of model. With the simulation code in which the dynamic behavior of a model is defined, the ability of SVG in *rube* 2D is extended from the model visualization capability to the model simulation capability.

3.2 Extensible 3D (X3D) Visualization and Modeling

The Naval Postgraduate School (NPS) performed research toward development of a scenario authoring and Web-based visualization capability [12]. The main activities are employing Web-based technologies for information content (XML) and 3D graphic content (X3D) to create an initial presentation of operations. The NPS project is called Scenario Authoring and Visualization for Advanced Graphical Environment (SAVAGE).

The main component for NPS's framework is composed of an authoring component and visualization component. The main feature of NPS's tool is automatic creation of a 3D model for an operation. The authoring component creates X3D for the scenario and the visualization component takes authorized X3D into VRML so the users can display their authoring.

3.3 Visual Simulation Environment: VSE (Balci)

The VSE is object-oriented, component-based, visual simulation model development and execution environment. Dynamic objects of model are entities which

moves physically or logically moves in a model, and graphically decomposed in a hierarchical and natural say in VSE.

The VSE solves problems if the problem can be simulated using discrete-event simulation [13]. The architecture of the VSE model consists of static architecture and the dynamic object, and a model component is either part of the model static architecture or a part of the dynamic object. The model structure is created in a hierarchical and natural manner. A standard built-in VSE class library is available to provide reusable model objects. Communication between objects is done by message passing.

The main processes of modeling and simulation in VSE are the editing model specification and simulation using a VSE simulator. A component in VSE can be created as a part of the model static architecture in the VSE editor. The VSE editor also translates the component created into executable code automatically. The VSE simulator provides an environment for execution, animation, and experimentation of the model created by the VSE editor.

The VSE is an object-oriented simulation environment. One of the benefits of VSE comes from component-based visual simulation and the capability of reusable simulation. These characteristics of VSE enable developers to observe large-scale visual simulation with reduced cost.

*rube*2D is differentiated from VSE in terms of the methodology of modeling and simulation. While VSE mainly focuses on object-oriented, component-based modeling and simulation with reusability, *rube*2D is mainly XML-based. *rube*2D separates model semantic from presentation and this gives freedom to the users to choose their own

metaphor for the model and a aesthetic aspect of modeling can be integrated with model creation.

3.4 Diagrammatic Representation and Reasoning

Diagrams are effective not only for representation but can also be useful to carry out a certain type of reasoning. A vocabulary definition of diagrammatic is a visual means of representation of information. Diagrams can be used as representational systems [14]. Diagrams are externally drawn representation systems rather than internal mental representations. Diagrammatic representations may use graphical element, as well as pictorial elements.

Recently, computer scientists, philosophers, and researchers have focused on diagrammatic representation systems. The interest in the diagrams has been generated by the fact that diagrams are more effective than other types of representations for certain types of tasks such as a reasoning process model, data and knowledge model, and scientific model. Zenon Kulpa summarized the main criteria characterizing the usefulness of a diagrammatic language as expressiveness, effectiveness, and presentation ability [15]. Euler's and Venn's diagrams are examples of how a mathematician's simple and internal intuition about diagramming reasoning has gradually been developed into an external formal representation system.

The diagrammatic representational models may describe a physical situation or abstract phenomenon. Diagrammatic representations of physical situation can be achieved by simplifying the physical objects or dimensions while abstract phenomenon can be presented by abstract element and relations through metaphor. Although diagrams do not necessarily need to be in 2D, many applications of diagrams are presented in 2D. That is probably because the aim of diagrammatic representation, effective reasoning, is

best achieved using 2D graphics. Of course, diagrams can be present in 3D. Modelers who want to model a diagrammatic representational system will find *rube*2D is a very qualified framework to achieve their modeling and simulation goal.

3.5 Sodipodi

Many kinds of free SVG editors are currently available on the Web. Sodipodi [16] is one of the most powerful and well-equipped free SVG editors. Sodipodi is an open-source, vector-based drawing program. It uses W3C SVG as its naïve file format. Thus model or graphic designers can create and visualize the SVG without any exporting or transformation, as most of free SVG editors require. Also Sodipodi supports a wide variety of flat form. The first version of Sodipodi was developed for Linux environment but the Windows version of Sodipodi is available with the All-in-One auto installer currently.

The main author of Sodipodi is Lauris Kaplinski. However, since Sodipodi is open-source free software, many people are participating in the development of Sodipodi and heading toward the construction of more reliable and efficient SVG editors together.

*rube*2D provides an extensive way of using Sodipodi in system modeling. Since SVG generated from Sodipodi is not compatible with *rube* 2D adaptable SVG, several processes are added in *rube*2D framework to make extensive use of Sodipodi SVG. Chapter 5 discusses this issue in more detail.

CHAPTER 4
RUBE 2D FRAMEWORK

In this chapter, diagrammatic dynamic modeling and simulation framework *rube*2D are discussed in detail. The paradigm and methodology of *rube*2D are introduced in the first section followed by the development environment in the second section. In Section 3, the overall structure of *rube*2D framework is described in detail. Finally, contributions of *rube*2D are discussed in Section 4.

4.1 Paradigm and Methodology

*rube*2D is an XML-based software modeling and simulation framework for 2D. *rube*2D is a ramification of *rube. rube*2D extends the capability of *rube* into the model domain of 2D while preserving the paradigm and the methodology [6]. The main idea of *rube*, that is, separating the topology of the model from its presentation, gives flexibility to the presentation of the model. That is to say, separating the topology from its presentation makes applying same topology over different model presentations possible.

Separation of the topology from the presentation gives freedom to model designers with not only the flexibility of applying the model topology over different 3D presentations or metaphor, but also making it possible to apply the model topology over 2D presentations of the model. If we can separate the model topology from its presentation and apply that topology over different 3D presentations, then we can easily think of the same mechanism with 2D presentations in *rube*. This gives an intuitive idea of *rube* 2D.

The power of *rube* 2D comes from the extensibility of *rube* modeling and simulation into model domains, which are more efficient and well described in 2D. The diagrammatic modeling and simulation for logical, mathematical, and physical systems in *rube* 2D enable developers to build more specific and precise models. Reasoning of the system in those fields can be performed with more care by concentrating the model semantic into model files in *rube* 2D. The diagrammatic representation of the model will also give a freedom to model developers to choose proper metaphors for their model to increase understandability along with the possibility of aesthetic modeling.

4.2 Development Environment

The primary development environment of *rube*2D is XML. The two components of the *rube,* model presentation file and model semantic file, are defined in XML. Also, every process of modeling, simulation, and visualization in *rube*2D, is associated with XML and XML technology.

In *rube*, the model presentation file is either in X3D or VRML. Since there is no existing X3D viewer, creating X3D document is cumbersome task. Developers can take alternative ways to compose the model presentation file with VRML, which will later be translated into X3D by available translation free tools. The final X3D with a simulation code also must be transformed into VRML to visualize the final output of simulation.

The release of XML for 2D graphics, SVG, has brought standardized and internationalized graphics in 2D graphics of the Web. *rube*2D is inspired by the interactivity and dynamics of SVG. By representing the model presentation file in SVG, *rube* framework accommodates XML in 2D. Unlike X3D, SVG is provided with many free viewers, which can be plugged into the Web browsers very easily. The SVG model

presentation file can be directly fed into the *rube*2D architecture without any transformation necessary for the model visualization.

*rube*2D provides GUI for MXL creation from model presentation in SVG. The model semantic, MXL, is a logical perception and understanding of the dynamic model about how it works while presentation, SVG, is physical perception and configuration of the model about how it looks. Both MXL and SVG in *rube*2D is different aspect of the same model and they are associated with each other. Logical representation of the model, the model semantic, can be extracted from the physical presentation. Likewise, the presentation of the model can be constructed from model semantics with extra information. The extra information involves configurations of model objects, as well as the layout of the model objects. In *rube*2D, by providing a way to construct a model semantic from a model presentation, developers can construct a model file from a presentation file. The GUI creating MXL from SVG is discussed in detail in Chapter 5.

4.3 Framework

The overall architecture of *rube* 2D is described in this section. The basic structure of *rube* 2D is almost identical to *rube* described in Chapter 2. The only difference is the language used in the model presentation file, SVG. The model representation in *rube*2D is achieved by decomposition of the dynamic model into two components: model presentation and model semantic. The model presentation of a dynamic model is defined in a presentation file written in SVG. The model semantic of a dynamic model is described in scene file written in MXL. The final SVG is created by internal transformation processes in *rube*2D architecture.

The SVG file, input to the *rube*2D, is purely static and has only geometry information of the model. However, every element in SVG can be easily integrated with

the rest of the architecture of *rube*2D.

The model file written in MXL is like an abstract and a high-level description of heterogeneous model semantics providing the model semantic of dynamic behavior. The model simulation code is generated transformations of the model file. First, MXL file is translated into a low-level model file, Dynamic eXchange Language (DXL). DXL is an assembly-level modeling language which represents a multimodel containing more than one heterogeneous sub-model type within homogeneous block diagrams. That is to say, heterogeneous model types described in MXL are transformed into homogeneous block diagrams model representation in DXL using Java DOM and SimPackJ/S [10]. Next, DXL is translated into executable simulation JavaScript codes. Finally, a Model Fusion Engine fuses the JavaScript codes with the SVG model presentation file. The output from the Model Fusion Engine is SVG file with associated simulation codes. The overall structure of *rube* 2D is shown in Figure 4-1.

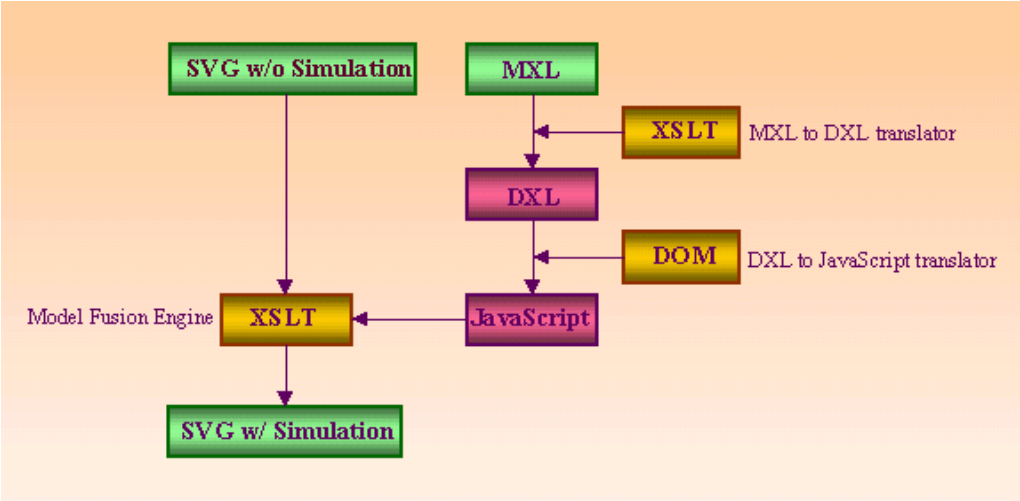

Figure 4-1. Overall structure of *rube*2D

The model fusion engine in *rube*2D is defined in XSLT to integrate SVG with the JavaScript simulation code. This new model fusion engine performs the same

functionality as it works in *rube*. The model fusion engine in *rube*2D will be discussed in Chapter 6 in detail.

To facilitate users with the creation of model presentation in SVG, *rube*2D provides a way to use an open-source SVG editor, Sodipodi [16], by providing an XSLT stylesheet which translates SVG generated by Sodipodi into *rube*2D applicable SVG. This stylesheet transform sodipodi SVG into *rube*2D framework accessible SVG by mainly reformatting by way of declaring attributes in SVG elements since the SVG element generated from Sopipodi uses a client-side element style definition, which cannot directly fit into *rube*2D.

4.4 *rube*2D Contribution

In this section, the main contributions of *rube*2D are discussed. Since *rube*2D is XML based, it takes strong power that XML technologies provide. *rube*2D extends the methodology of *rube*, and enables model developers to achieve personalized and aesthetic computing approach in modeling.

4.4.1 Open Source

The open-source software/tool is a software/tool software for which the source code is fully distributed or available with complied form without charge or limitations on modifications. No one is restricted from making use of the program in a specific field of endeavor [17]. The main advantages of the open source software are performance and reliability. According to the nature of an open source, the performance and reliability of the software can be easily proved from the fields. Linux, Apache Web Server, and Mozilla Web Browser are good examples of open-source projects.

*rube*2D framework is an open-source tool. It is available as a source code, and anybody can copy or modify the source code without any restriction. Because the main

development environment *rube* is XML and HTML, *rube* 2D can be easily edited or shared among the modeling and simulation community.

4.4.2 XML-Based Modeling and Simulation

One of the most prominent features of *rube* framework [6] is that it is XML-based. As a ramification of *rube*, *rube*2D is also founded on XML technology [1-5]. As mentioned in Chapter 2, XML is a Web standard for information exchange and repositories. The range of an XML application is extensive from e-business to mathematics applications. XML separates data from presentation. Separating information from details in the way it is to be represented brought huge advantages to the Web-based technologies.

In particularly, *rube*2D extends the feature of XML, separating data from the presentation, to the modeling and simulation domain. The model in modeling and simulation can be viewed from two different aspects. One aspect is from the view concerning the model semantic, which describes how the model is supposed to act. The other aspect is the model presentation, which describes the appearance of the model. The basic line of *rube*2D is separating model semantic from presentation as XML does. Both the model semantic and model presentation are written in XML in *rube*2D system. MXL is used to define the model semantic, and SVG describes the model presentation. Both MXL and SVG are XML sublanguages.

XML is a Web standard for information exchange and repositories. Thus XML is straightforwardly usable over the Internet. Web-based modeling and simulation promote publication and standardization of digital model objects, as well as the reusability of those objects. By taking this feature of XML, *rube*2D extends its power to the Web-based modeling and simulation.

The XML documents are easy to create, and it is easy to write programs which process XML documents. By building on cleanly implemented specifications, creation of XML-based implementations does not require a huge effort. XML also supports a wide variety of applications.

4.4.3 *rube* Methodology

*rube*2D methodology succeeds the *rube* methodology. The goal of *rube* is to create a model design methodology that supports a separation of a dynamic model semantic from presentation and visualization. In this way, creating more a personalized and aesthetic presentation of a model can be achieved with relatively less effort.

Separating model semantics from model presentations gives model developers a freedom to apply different types of model presentations or metaphors to the model semantics [18]. Model developers can construct their own meaningful model metaphor for model presentations as well as they can reuse any existing presentations. An aesthetic computing approach of modeling is also possible in *rube*2D modeling and simulation. The aesthetic computing approach in *rube*2D will be discussed in Section 4.4.4 in detail. The fact that visualizations can be highly customized by the user makes *rube* and *rube*2D unique from similar works.

4.4.4 Aesthetic Computing Approach Modeling

One of the primary purposes of *rube* and *rube*2D is to facilitate aesthetic formal models, especially mathematical and computing models. As prototyping machines and computer hardware power grows rapidly, representations of mathematics, sciences, and software have broaden their expressive capabilities. With increased efficiency for creating computing models, we have an opportunity to re-phrase formal representations of such model [19].

Representation of computational models can be achieved by means of metaphor. The use of metaphors can provide a new way of extending the existing computational model representation into a more exploratory, aesthetic medium. The model type which has a computational topology but does not have physical representation can be expressed in amore aesthetic way by exploring the use of artistic methods within common representations in computing.

Modelers who want to model an abstract scientific model into a more effective and expressive aesthetic form can achieve their goal in modeling and simulation with *rube*2D.

4.4.5 Integration of Model Presentation with Dynamic Model

A dynamic model is used to express and model the behavior of a system over time. It plays an important role in describing system behavior explicitly for system development, especially for a system design phase in software engineering. However, a dynamic model is symbolic presentation of system behavior, not actual presentation the system.

In a modeling and simulation world, visualization of a model is an essential part. While a dynamic model describes the behavior of a system over time, visualization of a model is a way to describe static presentation of the system. A dynamic model and model presentation are two different ways of understanding and describing the system.

In *rube*2D, users are able to juxtapose a dynamic model with a model presentation within the scene file. Both a dynamic model and model presentation can be simulated using the JavaScript simulation code result to represent the dynamic of the system and the picture of the model over time. By simulating a juxtaposed scene of a dynamic model with a model presentation, an understanding of the system is much easier.

CHAPTER 5
MODELING IN *RUBE* 2D

*rube*2D is a framework primarily for modeling and execution. In general, modeling is a way of reasoning about systems which involve the level of understanding of the system. Modeling or model design can be divided into two parts. One part is defining the dynamics of the model. The other part is defining the static presentation of the model. In this chapter, modeling in *rube*2D, constructing the model topology in MXL, and the creating model presentation in SVG is discussed in detail.

5.1 Model Representation in MXL

5.1.1 MXL Model Structure

The model file defines the topology and dynamic behaviors of the model type. The model file is defined in Multimodel eXchange Language (MXL). MXL provides an effective way to represent components and dynamics of the model, as well as simulation information in XML. To provide a concrete description of rules and syntax of the model file, which contains a topology of the model, XML schema is used. The schema of MXL is provided in the Appendix A.

Basic elements of the model file are ***model*** and ***simulation*** elements. The ***type*** attribute of the ***model*** element defines the type of model. Currently, the Functional Block Model (FBM) and Finite State Machine (FSM) are supported in *rube*2D. The ***topology*** element defines interconnectivity of model objects. The model object can be represented by a ***node*** element. The connectivity among model objects can be represented by an ***edge*** element. Each ***node*** or ***edge*** element has a ***script*** element, which contains a link to the

JavaScript file describing the dynamic behavior of the ***node*** or ***edge*** element. The ***simulation*** element contains information needed for the simulation of the model defined.

For the modeling example, we will use a simple equation in Eq. 5-1.

sin(x) *2 (Eq. 5-1)

The block diagram of the equation 5-1 with an additional print block for the equation is shown in Figure 5-1. First block is sine value generator. Second block multiplies it by two. The MXL definition for this FBM is given in Figure 5-2

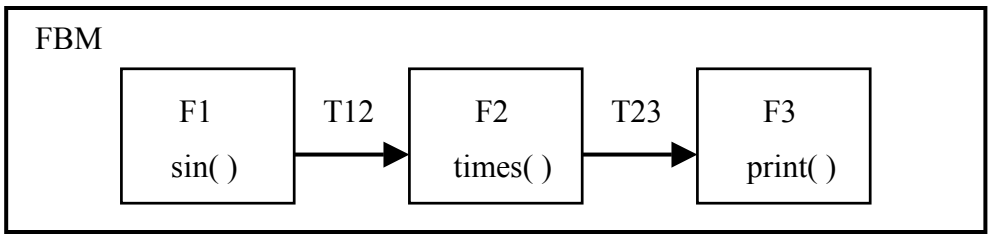

Figure 5-1. An FBM with three blocks

```
<?xml version="1.0" encoding="utf-8"?>
<MXL>
  <model type="FBM">
    <topology type="GRAPH">
      <node id="F1" type="FUNCTION" start="TRUE">
        <script id="sine.js" func="sin_func"/>
      </node>
      <node id="F2" type="FUNCTION">
        <script id="add.js" func="times_func"/>
      </node>
      <node id="F3" type="FUNCTION">
        <script id="print.js" func="print_func"/>
      </node>
      <edge id="T23" type="TRACE" begin="F2" end="F3" data_type="FLOAT">
        <script id=" "/></edge>
      <edge id="T12" type="TRACE" begin="F1" end="F2" data_type="FLOAT">
        <script id=" "/></edge>
      <edge id="T3X" type="TRACE" begin="F3" end="EX" data_type="FLOAT">
        <script id="" func=""/>
      </edge>
    </topology>
  </model>
<simulation start_time="0.0" end_time="20.0" delta_time="0.01" cycle_time="0.01" />
</MXL>
```

Figure 5-2. Model file for FBM with 3 blocks

The MXL in Figure 5-2 defines the dynamic model, FBM with three blocks, as described in Figure 5-2. The outermost element of MXL is ***MXL***. It has ***model*** and ***simulation*** elements as subelements.

The ***model*** element defines the type of the model. The model type is defined in ***type*** attribute of the ***model*** element. The topology of the model is described inside of the ***topology*** element. Since FBM is the directed graph model, the value ***GRAPH*** is given for the topology ***type*** attribute. Each ***node*** element describes each block of FBM with ***id, type*** and ***start*** attributes. The attribute ***id*** defines the name of the ***node*** element. The ***FUNCTION*** is the fixed node type for FBM MXL. The attribute ***start*** is added with a value of true when the node describes a starting or value generating block. The ***Edge*** element defines the interconnectivities between the ***node*** elements. It has attributes of ***id***, ***type***, ***begin***, ***end***, and ***data type***. As in the ***node*** element, attribute ***id*** defines the name of the ***edge*** element. The ***TRACE*** is the fixed edge type for FBM MXL. The ***begin*** and ***end*** attributes define the begin and end node ***id***s. The type of data interchanging between two blocks is defined by ***data_type*** attribute.

The behavior of model objects defined in each ***node*** or ***edge*** element must be specified in the separate JavaScript file. The ***id*** attribute of the ***script*** element refers to the JavaScript file that contains the behavior function listed in the ***func*** attribute. Details about structure rules, parameter, and variable naming rules must be followed for the JavaScript file and is discussed again in Section 5.1.2.

The ***simulation*** element contains information needed for the simulation of the model described inside of the ***model*** and ***topology*** elements. The ***simulation*** element has attributes that specify start time, end time, step size, and cycle delay needed for the model

simulation. Start time, end time, step size, and cycle delay of the simulation are defined in the attributes of ***start_time***, ***end_time***, ***delta_time***, and ***cycle_time***, respectively.

One additional edge with id="T3X" is defined in MXL in Figure 5-2, which does not exist in FBM in Figure 5-1. The edge "T3X" is defined to provide the output of the third block to the external object or model. This external edge is very useful to provide the model output to the outside world. It also modularizes the model as one object or function so this whole model can be placed within a multimodel environment as a submodel. Figure 5-3 shows a multimodel example of which FBM is nested inside of outer FBM.

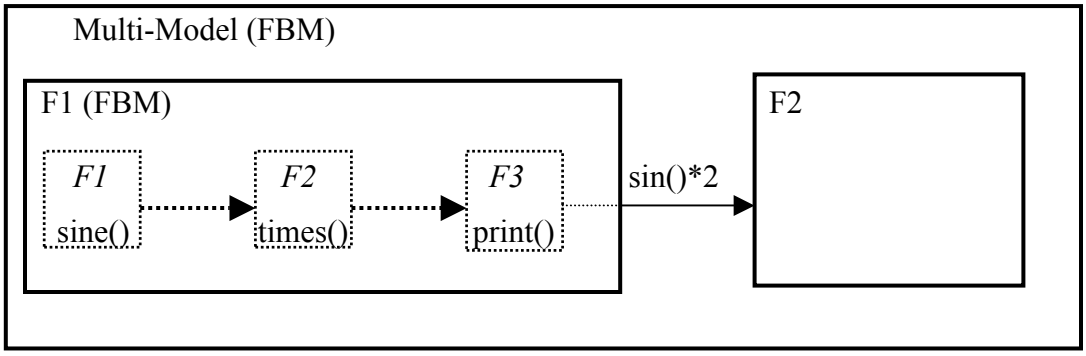

Figure 5-3. Multimodel example

5.1.2 JavaScript Functionality

JavaScript functions can be located in separated files. The location of a JavaScript file and function for the model object is referenced by ***id*** and ***func*** attributes of a ***script*** element of the model object. Every JavaScript function that interfaces with an MXL node or edge element takes no input parameters. Instead, a variable name of input to and output from the current function is fixed. Input to the current function can be specified by **this.<sourceID>,** and output to the target function from current function can be specified by **this.<targetID>**. The name of input to and output from the current function is fixed to accommodate the rest of the JavaScript simulation code, which will be obtained later via

MXL to DXL translation and DXL to JavaScript translation. Figure 5-4 shows an example of JavaScript functions for FBM MXL, as defined in Figure 5-2.

```
var dx = 0.1;
var x = 0;
function sine()
{
        this.F2 = Math.sin(x);
        x += dx;
}
function times()
{
        this.F3 = this.F1 * 2;
}
function print()
{
        out_F3 = this.F3;
}
```

Figure 5-4. JavaScript functions for the FBM MXL with three blocks

In the example JavaScript functions above, **this.<objectID>** used in the right-hand side is the input value to the current function and **this.<objectID>** used in the left-hand side is the output to the object with specified ID.

5.2 Model Representation in SVG

The modeler using *rube*2D can represent the model presentation in Scalar Vector Graphics (SVG) [5]. The SVG scene contains information that describes the physical appearance of the system. SVG can be created by hand from scratch using SVG syntax or by using SVG editors, such as Sodipodi [16]. By writing the scene file in SVG, modelers are able to describe the system in any way that they want to represent the system. The system what the modeler wants to represent may be a computational science system, virtual reality system, physically based system, or computer animation. Representing some of these systems might be associated with the physical appearance of the system

while the others might be associated with the perceptual understanding or artificial forms of the system.

Equation 5-1 is an example of science models which does not have a physical appearance associated with it. These types of model can be visualized in more abstract way or aesthetic way then the model which has a physical appearance. We will see two types of model representation for the equation model.

First, Figure 5-5 represents one way of SVG representation for the equation with FBM of three blocks.

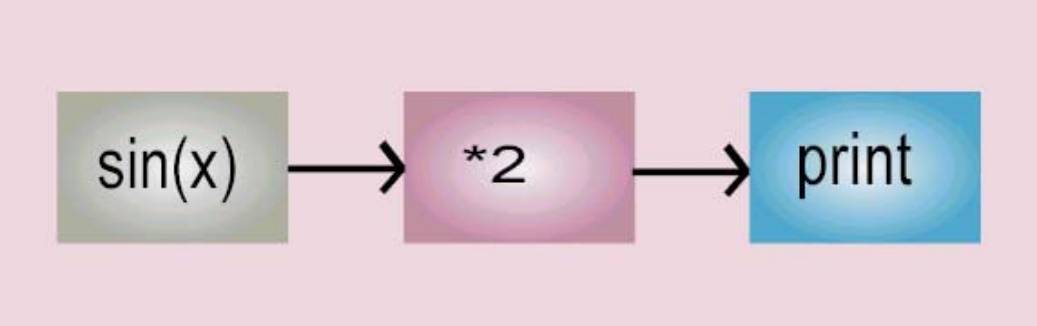

Figure 5-5. An SVG representation of FBM with three blocks

The model in Figure 5-5 describes the intuitive dynamic model presentation of FBM with three blocks. The model presented in this figure helps to understand the dynamic behavior of the system and suitable to help the understanding of the model behavior. The source code of a SVG in the Figure 5-5 is shown in Figure 5-6. Details about the SVG syntax and creation of SVG will be discussed in Section 5.3.

While the model presentation in Figure 5-5 gives intuitive understanding of the model semantics, we can construct another model presentations which are more like a physical system. The first picture in Figure 5-7 shows a possible way of representation of the FBM for equation model represented in Figure 5-5. It uses a graph and a blackboard metaphor to visualize the equation. The blue square inside of the graph represents the y

value of the equation. The values in the blackboard represent outputs from the first block and the second block. A more interesting presentation of the model is depicted in the second presentation. In the second figure of Figure 5-7, the dynamic model of FBM with three blocks is juxtaposed within the scene file.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20010904//EN"
"http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
<svg width="210mm" height="297mm" xmlns="http://www.w3.org/2000/svg">
    <rect style="fill:#ecd4dd;" id="box" x="37" y="141" width="574" height="182"/>
    <rect style="fill:#dd44ee;" id="sine" x="70" y="193" width="127" height="83" />
    <rect style="fill:#ddee44;" id="print" x="452" y="193" width="127" height="83"/>
    <rect style="fill:#44eedd;" id="add" x="262" y="193" width="127" height="83"/>
    <g id="Trace1" transform="matrix(0.999996,0,0,0.999996,6.28274,-87.9588)">
        <path style="stroke:black;stroke-width:3pt;"
          d="M 190.053 323.562 L 254.452 323.562 " id="path128"/>
        <path style="stroke:black;stroke-width:3pt;"
          d="M 254.452 323.562 L 243.457 336.128 " id="path129" />
        <path style="stroke:black;stroke-width:3pt;"
          d="M 254.452 323.562 L 245.028 312.567 " id="path130"/>
    </g>
    <g id="Trace2" transform="matrix(0.99999,0,0,0.99999,196.335,-86.388)">
        <path style="stroke:black;stroke-width:3pt;"
          d="M 190.053 323.562 L 254.452 323.562 " id="path136"/>
        <path style="stroke:black;stroke-width:3pt;"
          d="M 254.452 323.562 L 243.457 336.128 " id="path137" />
        <path style="stroke:black;stroke-width:3pt;"
          d="M 254.452 323.562 L 245.028 312.567 " id="path138"/>
    </g>
    <text style="fill:black;font-size:12px;" x="35" y="76"
        id="text157" transform="scale(2.63359,3.23986)">
        <tspan id="tspan158">sin(x)</tspan></text>
    <text style="fill:black;font-size:12px;" x="89" y="102"
        id="text160" transform="scale(3.27577,2.37369)">
        <tspan id="tspan161">+1</tspan></text>
    <text style="fill:black;font-size:12px;" x="176" y="72"
        id="text166" transform="scale(2.70968,3.35173)">
        <tspan id="tspan167">print</tspan></text>
</svg>
```

Figure 5-6. SVG source code for FBM

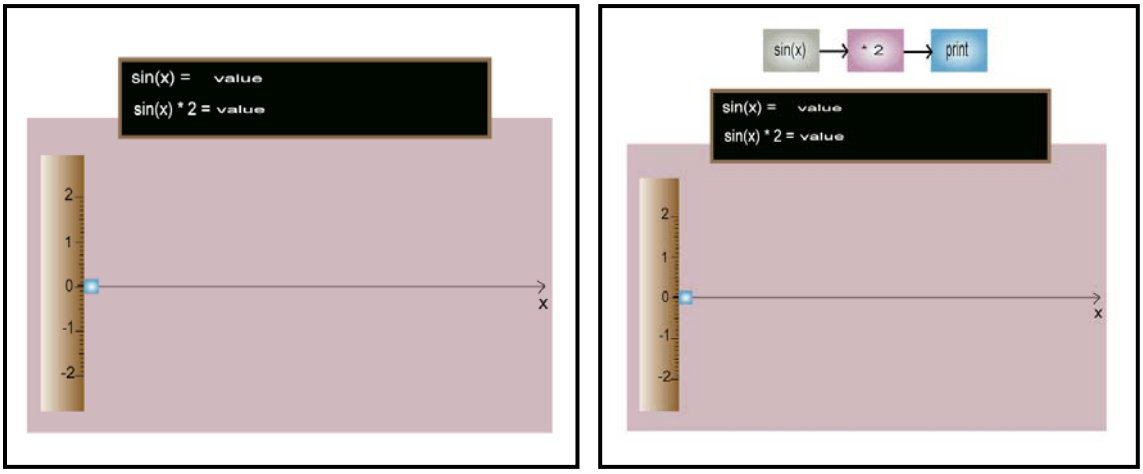

Figure 5-7. Different SVG representation of FBM with three blocks

Although the scene depicted in Figure 5-7 represents an abstract representation of the equation and is useful to help understand the dynamics of the system over time, it has no relation with how the real system should look.

When we model a system that has a physical appearance such as a physically based system or computer animation, modeling involves visualization of the system in a different way from the visualization of the abstract models. Visualization of such system can be achieved by representing the system in such a way that copies the physical appearance of the system using any graphical languages. Using metaphors for the model visualization might aid reasoning about the system. Most times the former representation is preferred to help understand how the system works or looks as we simulate.

Now we will take a look at the model representation of the system which has a physical appearance as in the example of a four-stroke gasoline or diesel engine. Fishwick [20] gives a detailed explanation of this model and its modeling and simulation.

The four-stroke engine has four phases or states, which repeat until the engine is turned off. The four phases continue in a cycle until the ignition is turned off. In the *compression* state, injected fuel vapor is compressed. After the fuel is compressed, it is

ignited in the *ignition* state. As a result of ignition, the cylinder is expanded in the *expansion* state. The resulting fumes exit through the tail pipe of the car in the *exhaustion* state. We can describe this four-stroke engine by the Finite State Machine (FSM). The FSM is depicted in Figure 5-8.

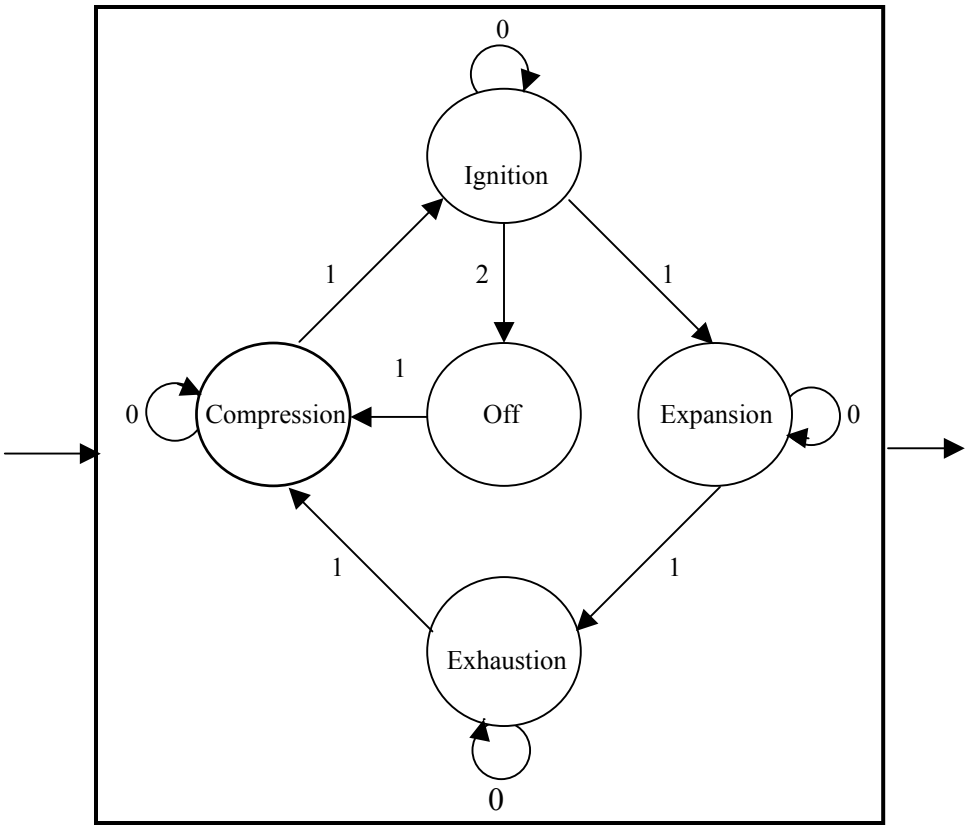

Figure 5-8. An FSM for a four-stroke engine

There are several ways to visualize the four-stroke engine in SVG. As one of example, we can build a model which describes the dynamic behavior, as given in Figure 5-8. The dynamic model representation of the four-stroke engine in SVG is presented in Figure 5-9.

We can present the four-stroke engine in a different way to describe physical appearance of each phase. Each of pictures in Figure 5-10 represents the state of an engine in each phase.

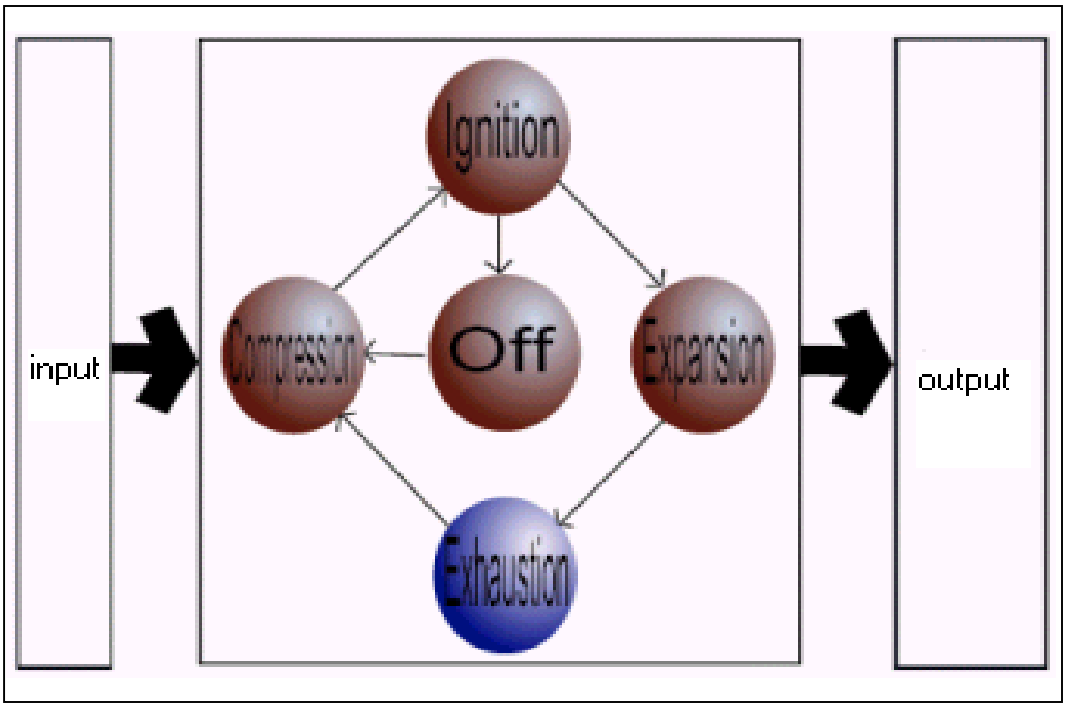

Figure 5-9. A SVG representation of a four-stroke engine dynamic model

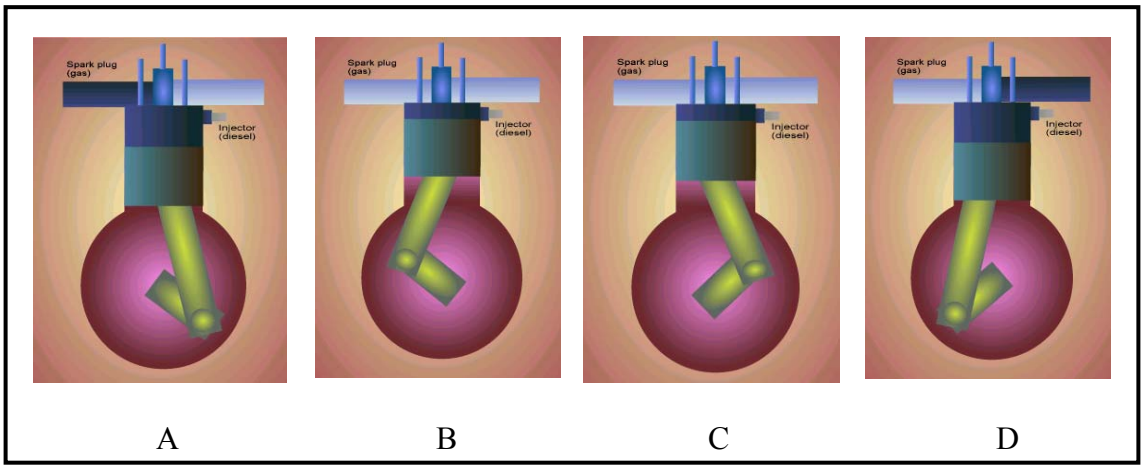

Figure 5-10. Phases of a four-stroke engine in SVG A) Intake stroke B) Compression stroke C) Power or work stroke D) Exhaust stroke

Models in Figure 5-9 and t Figure 5-10 are different representations of the same system. Although they are visualized in different ways, they have the same topology. Two different model representations can be simulated using a single model file defined in MXL within *rube*2D framework. With a fully defined MXL for the four-stroke engine,

users are able to simulate both model presentations using the JavaScript simulation code obtained from MXL. Details about modeling and simulation of the four-stroke engine will be introduced in Appendix B.

5.3 Model Creation

5.3.1 SVG Creation

SVG is text-based graphic language [5]. As with other XML, SVG can be created from any text editor. For textual formats, modeling is typically at the level of graphical objects rather than points. The three types of graphic objects in SVG are vector graphic shapes, image, and text. A general path element and basic shapes provided by SVG are fully qualified to construct models from simple worlds to very complex models. The basic shapes in SVG are the rectangle, circle, ellipse, line, polyline, and polygon. The filtering operation gives a raster effect on SVG drawings while the graphics are still scalable and displayable at different resolutions. Once developers understand how to construct SVG, they are free to build their own personalized model presentations.

SVG is relatively easy to create manually from text editors. However, having a visualized editor which creates SVG automatically from what we draw is very tempting. Various kinds of free SVG editors are currently available. Sodipodi is a recommended SVG editor in *rube*2D framework since its naïve file format is SVG, and, no exporting process is necessary to get an SVG format of drawing as the other editors require. In addition to that, Sodipodi supports the Linux system, as well as the Windows system. One of the nice features Sodipodi provides is the ‘XML editor’ functionality so users can modify attributes of SVG objects in a more sophisticated way from the ‘XML editor’ window. A snapshot of the SVG creating process using Sodipodi is shown in Figure 5-11.

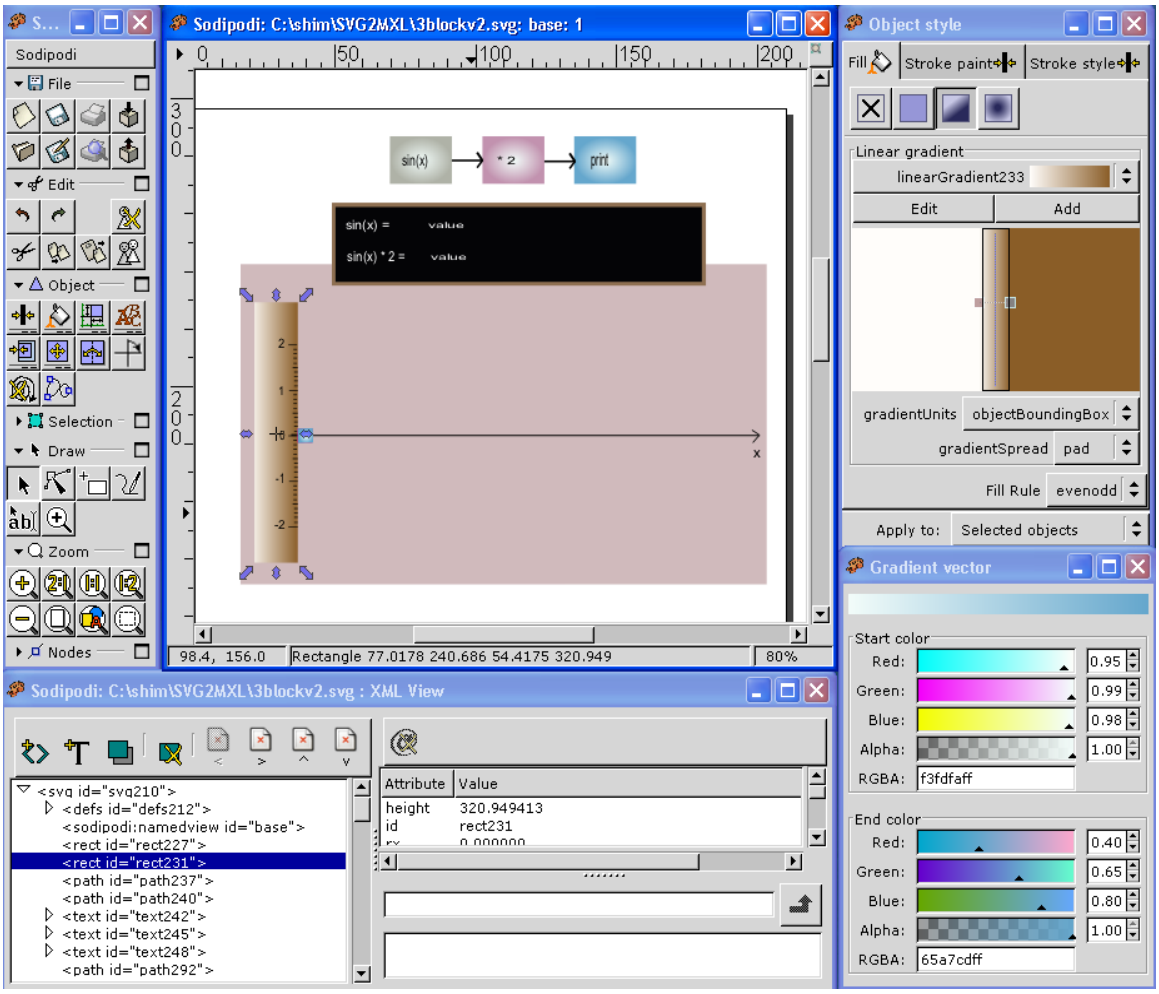

Figure 5-11. Snapshot of SVG creation using Sodipodi

5.3.2 MXL Creation from the SVG Model: GUI

The model presentation and model semantic provide a different understanding about the same model. The presentation describes a physical configuration of the model while the semantic describes the logical behavior of the model. We can extract the semantic of the model from the model presentation upon the understanding of how the model or system works. Likewise, we can construct the configuration of the model presentation from the model semantic if the geometric information of the model is available. The geometric information includes the physical appearances of each model objects and layout of those model objects.

The extracting presentation from the semantic would be a little more complex than the extracting semantic from the presentation. This is because constructing the presentation from the semantic involves other computer algorithms, such as graph layout algorithms. The output is highly dependent upon the algorithm used.

*rube*2D is facilitated with GUI, which produces the semantic of the model (MXL) from the presentation(SVG). With the help of GUI, users can create MXL from SVG without knowing the syntax of MXL. This prohibits any errors in modeling and simulation processes in *rube*2D caused by a malformed MXL document.

The GUI MXL creation tool is HTML-based. The user-created SVG is embedded inside of the HTML GUI. As the user selects the SVG element, the ID of that element is passed to the JavaScript in the HTML files. Users can also provide attribute values for the selected SVG element via interactive HTML elements, such as a textbox and a menu bar. The JavaScript inside of HTML files receives the ID of the SVG elements and user's input and then writes MXL with those values.

Characteristics of HTML and SVG used in the methodology of the MXL creation tool and GUI are as follows:

- SVG viewer can be plugged into a Web browser and SVG can be visualized in the Web browser
- SVG can be embedded within an HTML document
- HTML document can have SVG as a child document
- Both HTML and SVG are dynamic and interactive
- JavaScript in HTML can interact with SVG DOM
- JavaScript in SVG can make a call to external JavaScript functions defined out of SVG document

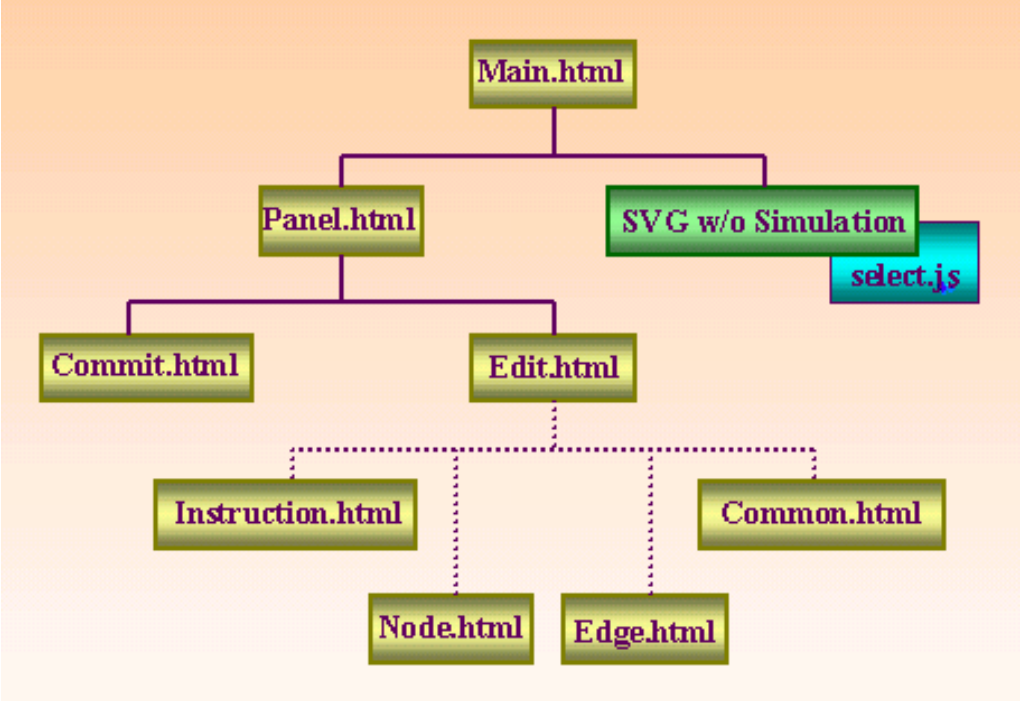

Figure 5-12. Framework of GUI of MXL creation tool

The main components in GUI are the SVG display section and MXL editing section. These two sections are embedded within the outer most HTML file. Figure 5-12 displays the overall structure of MXL creation tool with GUI.

The SVG displayed in the SVG section must be modified from the original SVG to response upon a user's selection on each of the SVG elements. In the modified SVG, every element is associated with the JavaScript function that changes the stroke of the selected SVG objects and hands over the SVG object ***id*** to the MXL creation section. As the user selects an object on the SVG section, the stroke color of the object is changed to notify the selection of the object. The ***id*** of the selected SVG object is displayed in the ***id*** textbox in the MXL editing section dynamically. XSLT stylesheet takes the original SVG and produces new SVG, which responds on a mouse click. The source code of the XSLT stylesheet for this transformation is provided in Appendix A. A newly generated interactive SVG uses the outside JavaScript (select.js). The JavaScript changes the stroke

color of the SVG elements and returns the value of the ID attributes. The XSLT stylesheet produces interactive SVG elements by setting the connection between the SVG elements and the JavaScript code.

The MXL editing section (panel.html) is subdivided into two frames again: one for editing section and one for the commend commit section. The commit section has buttons for viewing instruction, reset, file browsing, and MXL creation. The editing section is where the user inputs the values of the attributes about the selected SVG elements from the SVG section. The contents inside the editing section are changed dynamically according to the user's choice. As the user opens the GUI, the instruction of the usage of this tool is displayed in the editing section. When the user clicks on the '***Ready…***' button in the commit section, common.html is displayed in the editing section. The model type must be provided at this point by the user. The default model type is set to FBM.

Upon the user's selection of the element type, node or edge from the menu bar in the Common.html, the editing section is placed with node.html or edge.html, respectively. The *node.html* and *edge.html* are structured based on the MXL schema of the node and edge elements. Users can input the values for every attribute defined in node and edge elements from the editing section. The solid line in Figure 5-12 represents hierarchical relationship between files while the dashed line represents the sibling relationship.

The information the user gives on the selected SVG element is stored in the predefined data structure array in *panel.html*. When the user clicks on the 'Create' button in the *node.html* and *edge.html*, all the information the user inserted is stored in the data

structure. When the user selects the SVG element whose attributes are already defined, the JavaScript inside of the *panel.html* searches for the element array and retrieves the attribute values stored. The code fragment in Figure 5-13 is a JavaScript data structure for the SVG element.

```
function node
   (element,svg_id,id,type,n_data_type,start,begin,end,data_type,initial,script_id,func)
{
            this.element = element;
            this.svg_id = svg_id;
            this.id = id;
            this.type = type;
            this.n_data_type = n_data_type;
            this.start = start;
            this.begin = begin;
            this.end = end;
            this.data_type = data_type;
            this.initial = initial;
            this.script_id = script_id;
            this.func = func;
}
```

Figure 5-13. Data structure for SVG element in Select.js

A snapshot of the GUI of the MXL creation tool is shown in Figure 5-14. The MXL edition section is structured based on the MXL schema. The creation of the FBM MXL and FSM MXL is currently supported using the GUI introduced in this section. The creation of MXL for other types of models, such as the equation model and System Dynamics Model, will be equipped for this GUI functionality as research on *rube*2D progresses.

For the creation of erroneous MXL, there must be a way to catch all information needed for elements and attributes, as defined in the MXL schema for each model type. The GUI provides a way to define values upon the user's selection of the model type or element type for the set of elements or attributes defined in the MXL schema. The users are first required to choose the model type choice of the FBM MXL or FSM MXL

creation, and can define each ***node*** or ***edge*** elements by filling the textboxes fields or choosing from one of drop down menus. All the attributes for the ***simulation*** element in MXL can be defined.

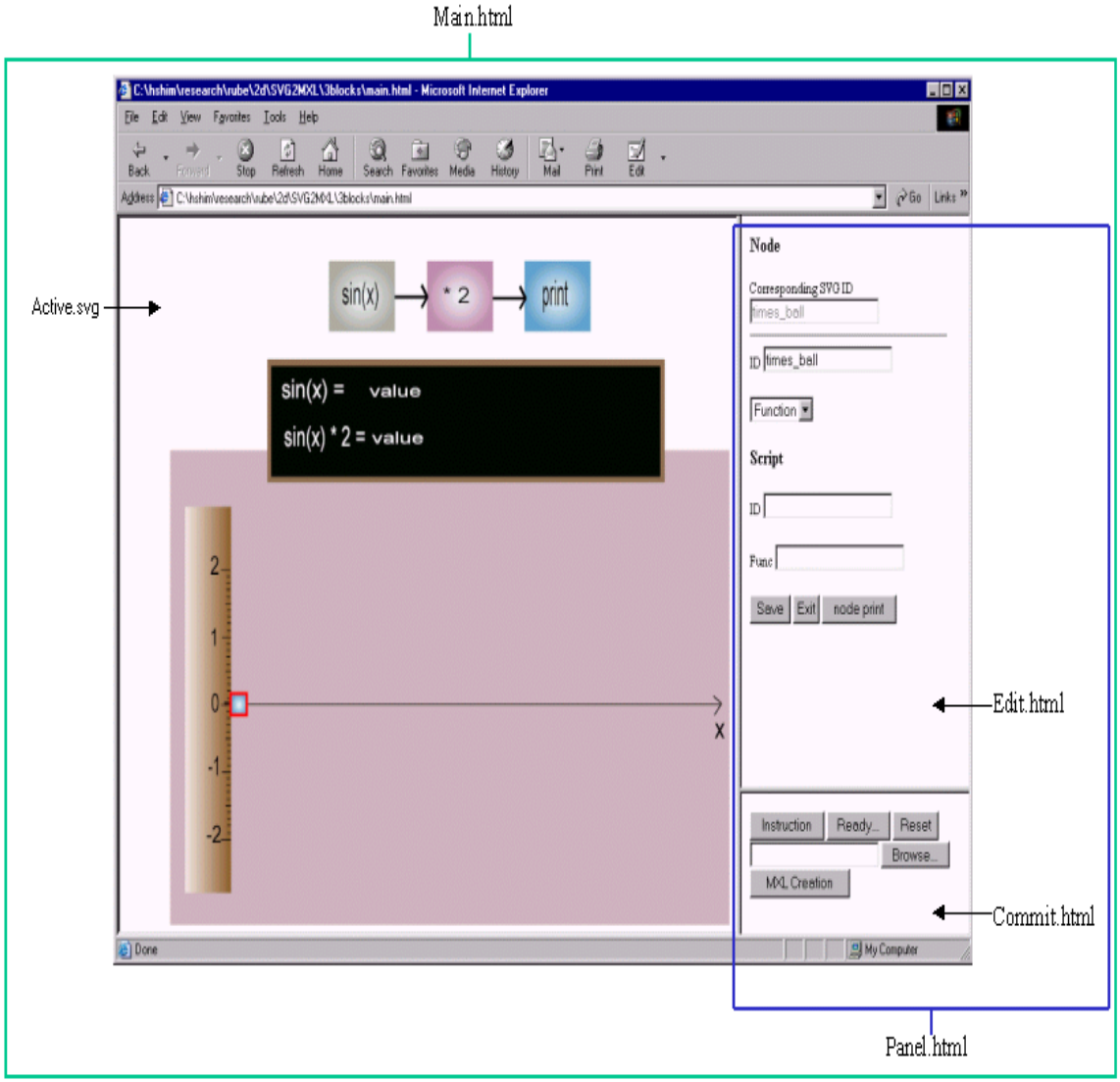

Figure 5-14. Snapshot of the MXL creation GUI

With the presence of the MXL creation tool with GUI, the extended structure of *rube*2D framework can be reconstructed. In the new structure of *rube*2D, users only need to construct a model presentation in SVG from scratch. The new version of rube2D architecture is depicted in Figure 5-14.

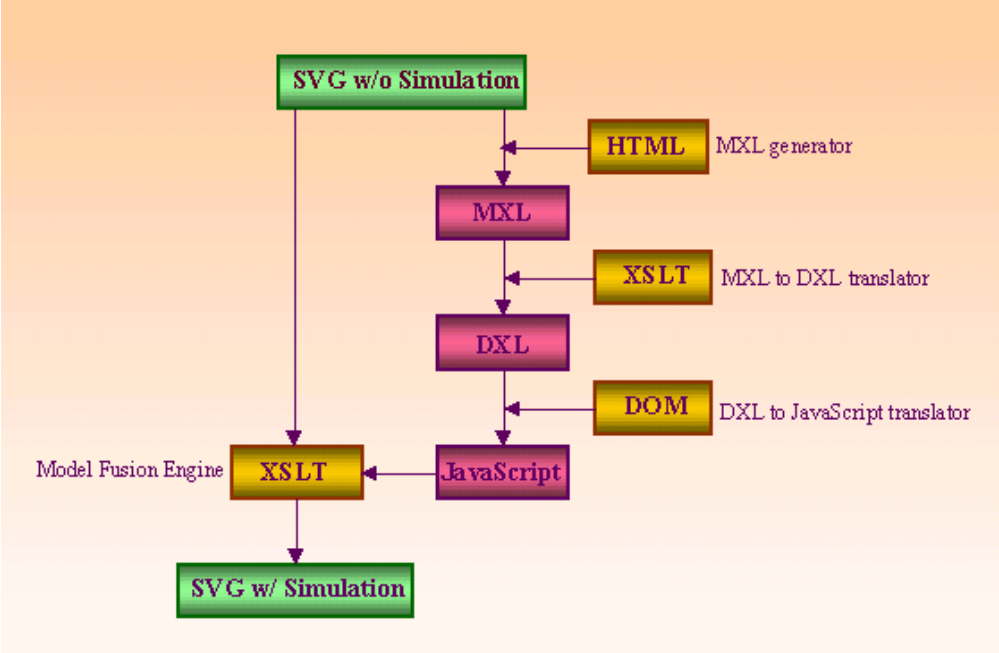

Figure 5-15. *rube*2D framework with the MXL creation GUI

CHAPTER 6
SIMULATION IN *RUBE* 2D

With a fully defined model file in MXL and the scene file in SVG, *rube*2D framework produces a simulation model. The model simulation in *rube*2D is done through several processes which mainly involve transforming the model topology defined in MXL into the JavaScript code and fusing the JavaScript code with the scene file defined by SVG. Each simulation step of *rube*2D is introduced in this chapter.

6.1 Overview

The simulation of the model defined in MXL and SVG separately is done to merge those two models into a single dynamic SVG which has the JavaScript code associated with it. The JavaScript code is obtained from MXL via two transformations inside of *rube*2D framework.

Inside of the transformation process from MXL to the JavaScript code, there exists an intermediate presentation of the model topology in Dynamic eXchange Language (DXL). The MXL is first transformed into DXL followed by DXL to the JavaScript transformation. The final JavaScript code is fused with the SVG scene file to provide a connection between them. Users can simulate the SVG scene file by using the JavaScript model execution code.

6.2 MXL to DXL Translation

While MXL has various elements for the heterogeneous model types, DXL has homogeneous model representation with block and connection. In other words, DXL represents heterogeneous model types within a homogeneous block model. The MXL to

DXL translation is done through the transformation using the rules defined in the XSLT stylesheet, MXL2DXL.xsl.

The DXL plays a role of the assembly layer between MXL and the executable JavaScript code. Every node element defined in MXL is translated into a block element in DXL while every edge element is translated into a connection element. While keeping the model topology as defined in MXL, DXL defines the model in a lower level. Figure 6-1 shows DXL translated from MXL introduced in Figure 5-2.

```
<?xml version="1.0" encoding="UTF-8"?>
<DXL xsi:schemaLocation="'dxl.xsd"'>
  <block id="F1" type="SYNC" >
        <port id="F1.OP1" type="OUTPUT" target="F2.IP1" data_type="FLOAT"/>
        <definition id="input.js" func="sine" start="TRUE"/>
  </block>
  <block id="F2" type="SYNC" >
        <port id="F2.IP1" type="INPUT" source="F1.OP1" data_type="FLOAT"/>
        <port id="F2.OP1" type="OUTPUT" target="F3.IP1" data_type="FLOAT"/>
        <definition id="input.js" func="times"/>
  </block>
  <block id="F3" type="SYNC" >
        <port id="F3.IP1" type="INPUT" source="F2.OP1" data_type="FLOAT"/>
        <port id="F3.OP1" type="OUTPUT" target="EX.IP1" data_type="FLOAT"/>
        <definition id="input.js" func="print"/>
  </block>
  <connect id="T12">
        <port id="T12.OP1" type="INPUT" source="F1.OP1" data_type="FLOAT"/>
        <port id="T12.IP1" type="OUTPUT" source="F2.IP1" data_type="FLOAT"/>
  </connect>
  <connect id="T23">
        <port id="T23.OP1" type="INPUT" source="F2.OP1" data_type="FLOAT"/>
        <port id="T23.IP1" type="OUTPUT" source="F3.IP1" data_type="FLOAT"/>
  </connect>
  <connect id="T3X">
        <port id="T3X.OP1" type="INPUT" source="F3.OP1" data_type="FLOAT"/>
        <port id="T3X.IP1" type="OUTPUT" source="EX.IP1" data_type="FLOAT"/>
  </connect>
  <simulation start_time="0.0" end_time="20.0" delta_time="0.01" cycle_time="0.01"/>
</DXL>
```

Figure 6-1. A DXL of FBM with three blocks

DXL can be viewed as a directed graph of blocks with ports and connections between blocks. Each block in DXL is associated with one or more input ports and output ports. The ***block***, ***connection***, and ***port*** elements are associated with the ***id*** attribute. The ***type*** attribute of the ***block*** element specifies synchronous or asynchronous execution of the block. The current version of DXL is defined only to simulate blocks synchronously. The value of ***type*** attribute of the ***port*** element is either **INPUT** or **OUTPUT**. The ***source*** or ***target*** attribute of the ***port*** element specifies the source and target ports. The ***data_type*** attribute of the ***port*** element defines the data type following between two blocks. The data type of the output block port is matched with the data type of the input block port.

By simplifying the representation of the model topology, DXL can be easily translated into the JavaScript code. Figure 6-2 shows an FBM DXL with three blocks.

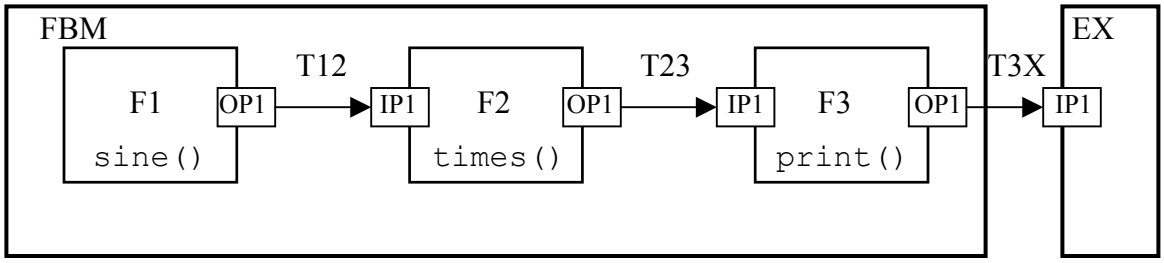

Figure 6-2. DXL block diagram for FBM with three blocks

6.3 DXL to JavaScript Translation

Generation of the final execution code, JavaScript, for the model simulation is done by DXL to the JavaScript translation using Java DOM. JavaScript code obtained from DXL, user input JavaScript function defined for the behavior of each block, and SimPackJ/S [10] are used together to perform the final simulation. SimPackJ/S is a set of simulation programs written in JavaScript. SimPackJ/S provides necessary functions, such as create_list(),schedule(), and next_event() for the final JavaScript code translated from DXL.

The final output JavaScript code consists of three parts: code copied from SimPackJ/S [10], code copied from user input script, and code generated based on the model topology. The DXL to the JavaScript Translator (DXL.class) copies the user input script functions and necessary functions from SimPackJ/S. The DXL to the JavaScript Translator glues the user input script and SimPackJ/S functions together and produces a simulation code which based on model topology. Figure 6-3 is the abstracted segmentation of the JavaScript code generated from DXL in Figure 6-1.

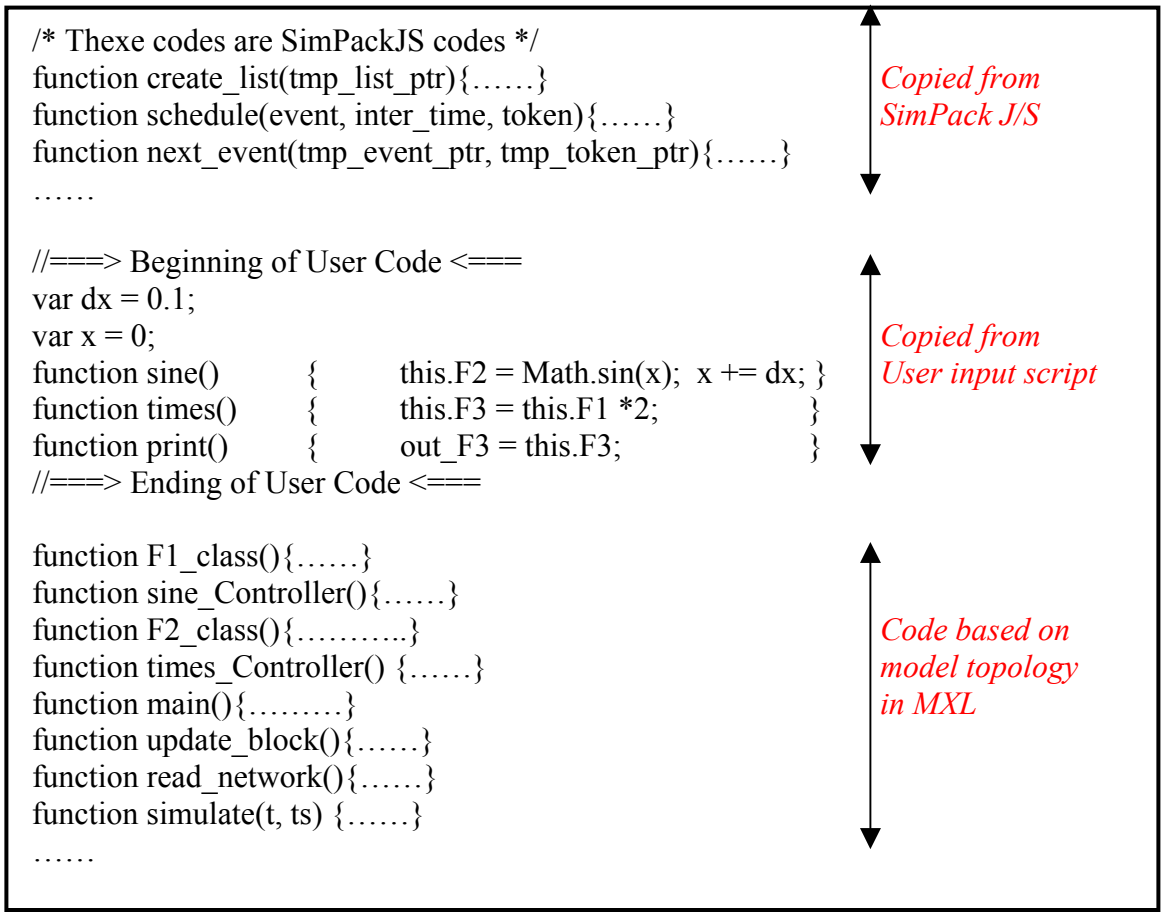

Figure 6-3. Abstract segmentation of the JavaScript code generated from FBM DXL

As a result of running the simulation code, we can produce the outputs from each model objects, ***node*** elements in MXL over time. The result of each ***node*** element is calculated and stored in the model variables inside of the ***updata_block()*** method. The

model variables have a name of ***out_<node_id>***. Figure 6-4 shows the code inside of the ***updata_block()*** method assigning the model simulation results to the model.

```
function update_block()
{
    switch (event.value)
    {
     case 0:……
            out_F1 = F1_object.out_ports[0];
            ……
            break;
     case 1:……
            out_F2 = F2_object.out_ports[0];
            ……
            break;
     case 2:……
            out_F3 = F3_object.out_ports[0];
            ……
            break;
     }
}
```

Figure 6-4. Abstract code about the model variables in simulation code

Although the final JavaScript code will be used to activate the SVG objects by model fusion engine, the final JavaScript code also can be used as a stand-alone code without a model presentation with a slight code addition: print statements of the model variables and calling to the main() method. To test the simulation result from the JavaScript code, users can embed the final JavaScript code inside of the <script> element of the HTML document or run the code from the JavaScript implementation tool such as Rhino [21].

6.4 Model Fusion Engine

The Model Fusion Engine is an XSLT-based process. It takes two XML files, SVG scene file and MXL model file, and produces an SVG simulation file. The final

SVG simulation file has exactly the same geometry information as the input SVG file with simulation JavaScript code included.

To simulate the SVG elements using the JavaScript code generated from MXL, several processes are required. First, a connection must exist between the SVG scene file and the simulation JavaScript code. Second, the output of the simulation code must be accessible from the script inside of SVG. And finally, simulation of the SVG scene file is done using the result of the simulation code and changing attribute values of the elements.

The Model Fusion Engine, which is written in XSLT stylesheet, defines rules of transforming from the original SVG into the merged SVG with MXL. The Model Fusion Engine performs the following tasks:

- Copies original SVG scene file
- Makes a link to the JavaScript code generated from MXL
- Declares variables for SVG elements
- Assigns the references of the SVG elements to the variables
- Declares parameters to the output variables of the JavaScript code
- Roads starting method, *main()*, and call simulation method, *simulation()*, in the simulation JavaScript code
- Adds simulation handle (HUD)

The final SVG is programmed with all requirements listed above. By using JavaScript code output parameters, users can manipulate the attributes of SVG elements. Figure 6-5 is a code segment from XSLT style sheet. It makes a link to the simulation JavaScript code and declares variables with the SVG element ***id***s and assigns SVG elements into declared variables.

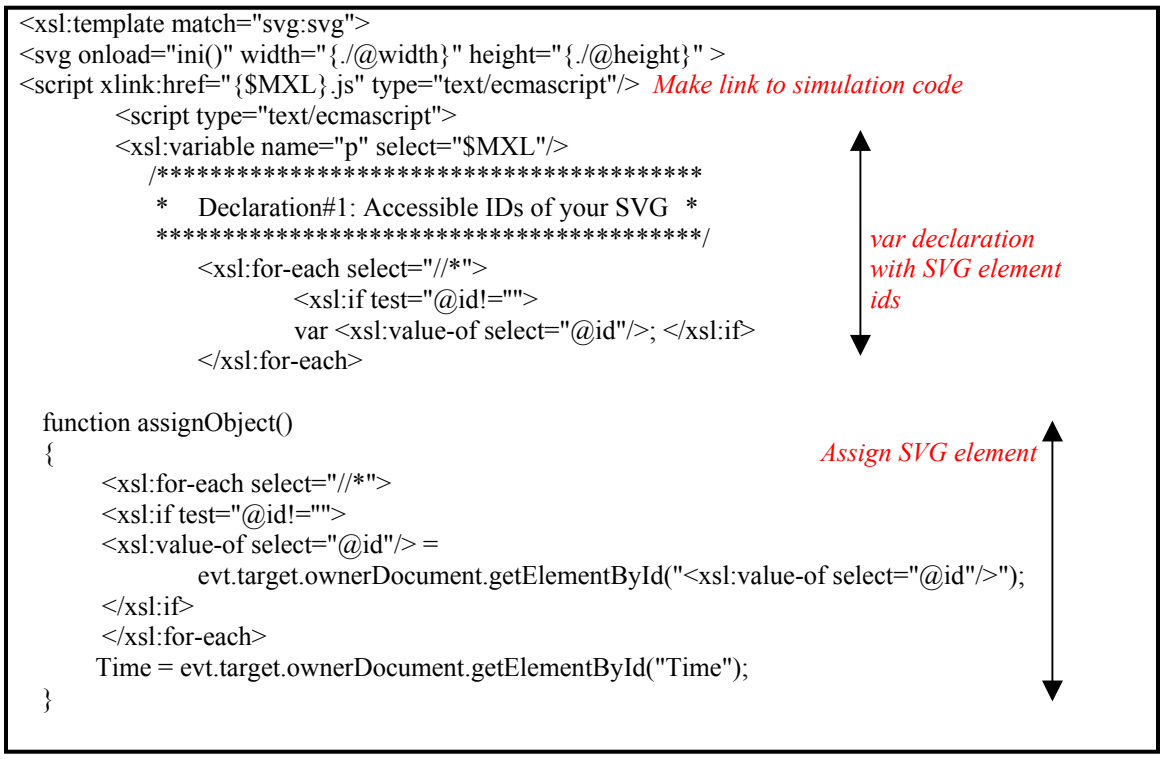

Figure 6-5. Code segment from the Model Fusion Engine (SVGmerge.xsl)

For theSVG document transforming using the XSLT stylesheet, we need an XSLT, processor such as Saxon [#2.2-2], Xalan, and Microsoft MSXML3. The principal role of an XSLT processor is to apply XSLT stylesheet to an XML source document and produce a result document. The XSLT processor takes the XSLT stylesheet and applies the rules over the original SVG file. We need to process two source documents, SVG and MXL, by a single XSLT stylesheet. The main source document is SVG. The file name of MXL will be returned to the XSLT stylesheet as a parameter from the XSLT processor command. XSLT stylesheet sets the connection with the simulation code and adds miscellaneous simulation code into the original SVG by reading MXL.

Figure 6-6 shows the segment of the final SVG generated from merging the model file in Figure 5-2 with the JavaScript code in Figure 5-4, and the scene file in Figure 5-7.

```
<svg onload="ini()" width="210mm" height="297mm">
  <script xlink:href="3f2t_mxl.js" type="text/ecmascript"/>  Link to the simulation code
  <script type="text/ecmascript"><![CDATA[
    var base;
    var sine;
    ……                                          Declare variables for SVG elements

    function assignObject()
    {       base = evt.target.ownerDocument.getElementById("base");
            sine = evt.target.ownerDocument.getElementById("sine");
     ……}                                        Assign SVG elements

    var out_F1
    var out_F2
    var out_F3                                  Parameters to the model variables in the simulation code

    function user_code()
    {}                                          User's function for SVG element simulation (left blank on purpose)

    function rube()
    {……}                                        Simulation setting method
    ……
  ]]></script>
<g>
……
<text onclick="rube()" x="10" y="40" font-size="15">Start</text>   Simulation handle (HUD)
<text x="10" y="60" font-size="15">Simulation Time:</text>
</g>

 ……  SVG elements copied from original SVG
</svg>
```

Figure 6-6. Code segment of the final SVG

The merged SVG is the final output of the *rube*2D framework. This SVG includes a precisely defined scripting code, which produces the results for each model object, as defined in the model topology file, MXL. Simulation of the merged SVG is done through changing the attribute values of the element. Details about the SVG element simulation will be discussed in Section 6.5.

6.5 Model Simulation

The JavaScript code produces outputs from each ***node*** element defined in MXL. The simulation outputs of ***node*** elements in MXL are calculated and stored in the model variable named ***out_<node_id>*** in the simulation code. Those values can be used to

analyze the performance of the model or to show the state of each model object over time. To actually use the output values for the simulation of SVG, we need to use the model variables in the simulation code to change the attribute values of the SVG elements.

Every ID of the SVG elements are declared as variables and the reference to the element is stored in its variables. Figure 6-7 is the code fraction from the merged SVG.

```
var sine_value;
var times2_value;
var sine_path;
var times2_path;
var sine_ball;
var times_ball;

function assignObject()
{
    sine_value = evt.target.ownerDocument.getElementById("sine_value");
    times2_value = evt.target.ownerDocument.getElementById("times2_value");
    sine_path = evt.target.ownerDocument.getElementById("sine_path");
    times2_path = evt.target.ownerDocument.getElementById("times2_path");
    sine_ball = evt.target.ownerDocument.getElementById("sine_ball");
    times_ball = evt.target.ownerDocument.getElementById("times_ball");
    Time = evt.target.ownerDocument.getElementById("Time");
}

var out_F1
var out_F2
var out_F3

function user_code()
{
    //User will fill this function later to manipulate SVG element
}
```

Figure 6-7. Code for the variable declaration and assign statements in the merged SVG

The function ***user_code()*** in Figure 6-5 has no code in the body. This is where users can use the model variables to simulate SVG elements in the way they want. Figure 6-8 shows the example of the modified ***user_code()*** method.

```
var sine_draw = "M 181.035 690 L 189.92 690 "

 var times_draw = "M 179.925 670.3 L 187.699 670.3 "
 var pre_out1 = 0;
 var pre_out2 = 0;

 function user_code()
 {
    board_sine_value.firstChild.data  = out_F1;
    board_times2_value.firstChild.data  = out_F2;

    sine_ball.setAttribute('y', 396 - out_F1*56);
    times_ball.setAttribute('y', 396 - out_F2*56);

    diff1 = pre_out1 - out_F1;
    diff2 = pre_out2 - out_F2;

    sine_draw = sine_draw + "l 0.5 " + diff1*55 + " ";
    sine_path.setAttribute('d', sine_draw);

    times_draw = times_draw + "l 0.5 " + diff2*55 + " ";
    times2_path.setAttribute('d', times_draw);

    pre_out1 = out_F1;
    pre_out2 = out_F2;
 }
```

Figure 6-8. Example of the modified ***user_code()*** method in the merged SVG

In the example code in Figure 6-6, the simulation result of the first block, ***out_F1***, is used to change the value of the text element '***board_sine_value***', change the position of the rectangle element '***sine_ball***', and draw an extended path from the path element '***sine_path***'. Likewise, the result of the second block, ***out_F2***, is used to change the value of the text element '***board_times2_value***', change the position of the rectangle element '***path_ball***', and draw an extended path from the path element '***times2_path***'. Figure 6-9 is a snapshot of running the merged SVG after adding the simulation codes in Figure 6-8.

The final output of *rube*2D is SVG that has the same geometry as the original SVG and dynamic behavior as defined in the model topology file, MXL. The simulation code

generated based on the model topology defined in MXL is used to simulate the elements in the merged SVG. Figure 6-9 shows the result of using the simulation code to draw graphs and text values.

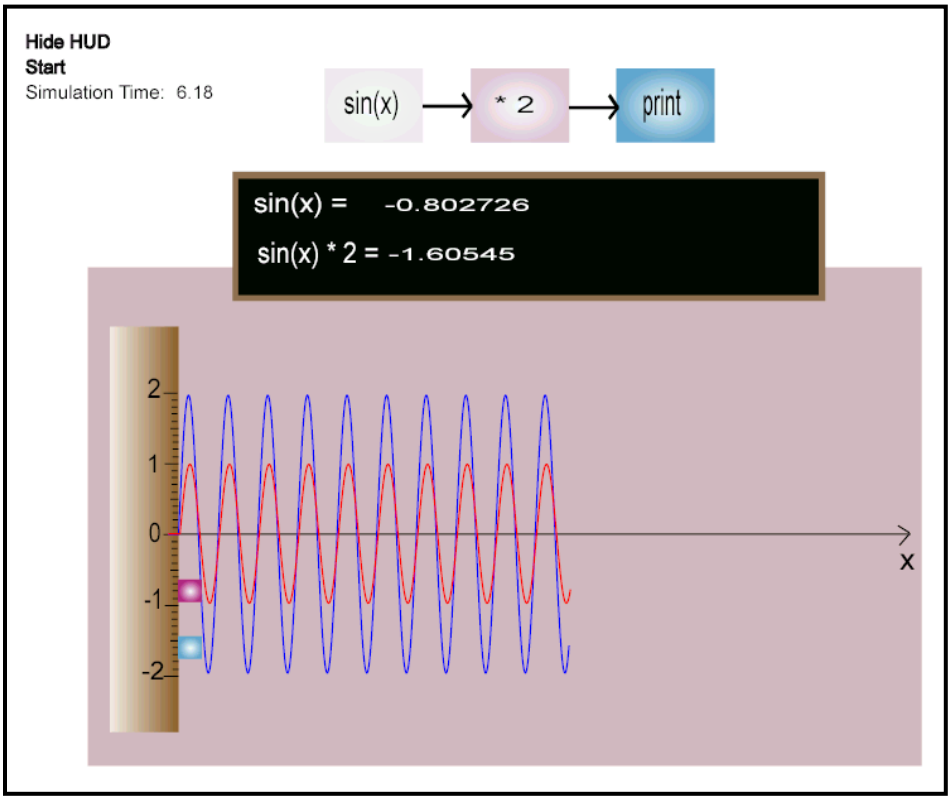

Figure 6-9. Snapshot of the final SVG

CHAPTER 7
CONCLUSION

My research goal is to provide a powerful and effective Web-based modeling and simulation environment, *rube*2D, for diagrammatic dynamic models. *rube*2D enables modelers to define model semantic and a model presentation, and acquire an automated simulation model from them. By separating the model topology from the presentation, *rube*2D provides a totally personalized dynamic model representation and extends the power of modeling and simulation with an aesthetic computing approach and integration of model presentation with a dynamic model.

The model design in *rube*2D is separated in two parts: topology design in Multimodel eXchange Language (MXL) and presentation design in SVG. By representing a model presentation in SVG, *rube*2D inherits all the advantages of XML technologies. The model presentation written in SVG can be easily accessed and modified or transformed. The model topology stored in MXL captures the dynamic behavior of the model over time. To facilitate modelers creating MXL, an automated MXL creation tool with GUI is provided in *rube*2D. This GUI lets modelers extract model topology from the presentation and construct MXL from it.

Simulation in *rube*2D involves several steps: generating simulation code from MXL and merge SVG with the simulation code. To obtain the JavaScript simulation code, model topology in MXL must be translated into a different and executable form. Before translated into the simulation code, MXL is first translated into Dynamics eXchange Language (DXL). DXL is an intermediate presentation of model topology.

With simplified model topology in DXL, translation into the simulation code is much easier. The Model Fusion Engine written in XSLT stylesheet merges the simulation code with SVG scene file to produce a dynamic SVG. Finally users can simulate the dynamic SVG by changing the attribute values of SVG elements using the JavaScript simulation code.

*rube*2D provides extensive modeling and simulation environment for modelers to achieve personalized and customized modeling and aesthetic computing modeling in 2D diagrammatic dynamic models. It also provides a convenient and modularized simulation while enables modelers the Web-based modeling and simulation with a new emerging XML technology.

*rube*2D supports only for Functional Block Model (FBM) and Finite State Machine (FSM) currently. The other model types will be supported by *rube* and *rube*2D as the research progresses.

APPENDIX A
SCHEMA FOR MXL AND DXL

```
<?xml version="1.0" encoding="utf-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
        Multimodel eXchange Language(MXL) schema for rube architecture.
        Copyright 2001 University of Florida. All rights reserved.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:element name="MXL" type="MXLType"/>
    <xsd:complexType name="MXLType">
        <xsd:sequence>
            <xsd:element name="model" type="modelType"/>
            <xsd:element name="simulation" type="simulationType"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="modelType">
        <xsd:sequence>
            <xsd:element name="topology" type="topologyType"/>
        </xsd:sequence>
        <xsd:attribute name="type" type="xsd:string"/>
    </xsd:complexType>
    <xsd:complexType name="topologyType">
        <xsd:sequence>
            <xsd:element name="node" type="nodeType" minOccurs="0" maxOccurs="unbounded"/>
            <xsd:element name="edge" type="edgeType" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="type" type="xsd:string"/>
    </xsd:complexType>
    <xsd:complexType name="nodeType">
        <xsd:attribute name="id" type="xsd:string"/>
        <xsd:attribute name="type" type="xsd:string"/>
        <xsd:attribute name="start" type="xsd:string"/>
    </xsd:complexType>
    <xsd:complexType name="edgeType">
        <xsd:attribute name="id" type="xsd:string"/>
        <xsd:attribute name="type" type="xsd:string"/>
        <xsd:attribute name="begin" type="xsd:string"/>
        <xsd:attribute name="end" type="xsd:string"/>
        <xsd:attribute name="data_type" type="xsd:string"/>
        <xsd:attribute name="initial" type="xsd:string"/>
    </xsd:complexType>
    <xsd: complexType type="simulationType"/>
        <xsd:sequence>
            <xsd:element name="simulation" type="simulationAttributeType"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="simulationAttributeType">
        <xsd:attribute name="start_time" type="xsd:string" use="required"/>
        <xsd:attribute name="end_time" type="xsd:string" use="required"/>
        <xsd:attribute name="delta_time" type="xsd:string" use="required"/>
        <xsd:attribute name="cycle_time" type="xsd:string" use="required"/>
    </xsd:complexType >
</xsd:schema>
```

Figure A-1. MXL schema

```
<?xml version="1.0" encoding="utf-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema" elementFormDefault="qualified">
    <xsd:element name="DXL">
        <xsd:annotation>
            <xsd:documentation>Comment describing your root element</xsd:documentation>
        </xsd:annotation>
        <xsd:complexType>
            <xsd:element name="block">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element name="port">
                            <xsd:attribute name="id" type="xsd:string" use="required"/>
                            <xsd:attribute name="type" type="xsd:string" use="required"/>
                            <xsd:attribute name="target" type="xsd:string" use="optional"/>
                            <xsd:attribute name="data_type" type="xsd:string" use="optional"/>
                        </xsd:element>
                        <xsd:element name="definition">
                            <xsd:attribute name="id" type="xsd:string" use="required"/>
                            <xsd:attribute name="func" type="xsd:string" use="required"/>
                            <xsd:attribute name="start " type="xsd:string" use="optional"/>
                        </xsd:element>
                    </xsd:sequence>
                </xsd:complexType>
                <xsd:attribute name="id" type="xsd:string" use="required"/>
            </xsd:element>

            <xsd:element name="connect">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element name="port">
                            <xsd:attribute name="id" type="xsd:string" use="required"/>
                            <xsd:attribute name="type" type="xsd:string" use="required"/>
                            <xsd:attribute name="source" type="xsd:string" use="optional"/>
                            <xsd:attribute name="data_type" type="xsd:string" use="optional"/>
                        </xsd:element>
                    </xsd:sequence>
                </xsd:complexType>
                <xsd:attribute name="id" type="xsd:string" use="required"/>
            </xsd:element>

            <xsd:element name="simulation">
                <xsd:attribute name="start_time" type="xsd:string" use="required"/>
                <xsd:attribute name="end_time" type="xsd:string" use="required"/>
                <xsd:attribute name="delta_time" type="xsd:string" use="required"/>
                <xsd:attribute name="cycle_time" type="xsd:string" use="required"/>
            </xsd:element>

        </xsd:complexType>
        <xsd:attribute name="id" use="required"/>
    </xsd:element>
</xsd:schema>
```

Figure A-2. DXL schema

APPENDIX B
MODELING AND SIMULATION EXAMPLE

Simple Model: Functional Block Model

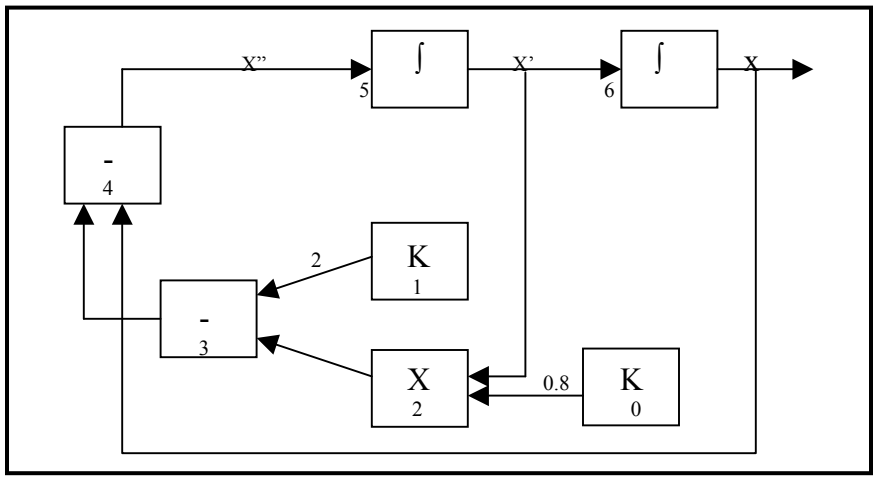

Figure B-1. Block diagram for x” + 0.8x’ + x = 2

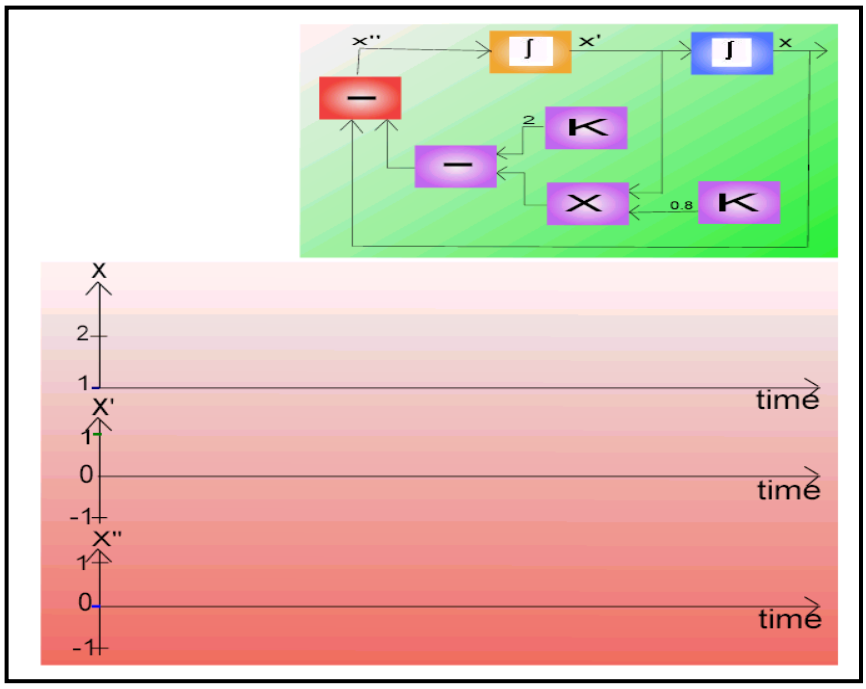

Figure B-2. SVG scene for the FBM in Figure B-1

```
<?xml version="1.0" encoding="utf-16"?>
<MXL>
 <model type="FBM">
                        <topology type="GRAPH">
                                <node id="F0" type="FUNCTION" start="TRUE">
                                        <script id="input.js" func="con1"/>
                                </node>
                                <node id="F1" type="FUNCTION" start="TRUE">
                                        <script id="input.js" func="con2"/>
                                </node>
                                <node id="F2" type="FUNCTION">
                                        <script id="input.js" func="mul"/>
                                </node>
                                <node id="F3" type="FUNCTION">
                                        <script id="input.js" func="sub1"/>
                                </node>
                                <node id="F4" type="FUNCTION">
                                        <script id="input.js" func="sub2"/>
                                </node>
                                <node id="F5" type="FUNCTION">
                                        <script id="input.js" func="int1"/>
                                </node>
                                <node id="F6" type="FUNCTION">
                                        <script id="input.js" func="int2"/>
                                </node>
                                <edge id="TR02" type="TRACE" begin="F0" end="F2" data_type="FLOAT">
                                        <script id="" func=""/>
                                </edge>
                                <edge id="TR13" type="TRACE" begin="F1" end="F3" data_type="FLOAT">
                                        <script id="" func=""/>
                                </edge>
                                <edge id="TR23" type="TRACE" begin="F2" end="F3" data_type="FLOAT">
                                        <script id="" func=""/>
                                </edge>
                                <edge id="TR34" type="TRACE" begin="F3" end="F4" data_type="FLOAT">
                                        <script id="" func=""/>
                                </edge>
                                <edge id="TR45" type="TRACE" begin="F4" end="F5" data_type="FLOAT">
                                        <script id="" func=""/>
                                </edge>
                                <edge id="TR52" type="TRACE" begin="F5" end="F2" data_type="FLOAT" initial="1.0">
                                        <script id="" func=""/>
                                </edge>
                                <edge id="TR56" type="TRACE" begin="F5" end="F6" data_type="FLOAT">
                                        <script id="" func=""/>
                                </edge>
                                <edge id="TR64" type="TRACE" begin="F6" end="F4" data_type="FLOAT" initial="1.0">
                                        <script id="" func=""/>
                                </edge>
                                <edge id="TR6X" type="TRACE" begin="F6" end="EX" data_type="FLOAT">
                                        <script id="" func=""/>
                                </edge>
                        </topology>
        </model>
        <simulation start_time="0.0" end_time="20.0" delta_time="0.01" cycle_time="0.01" />
</MXL>
```

Figure B-3. MXL for the FBM in Figure B-1

```
function con1(block_token)
{
        this.F2 = 0.8;
}

function con2(block_token)
{
        this.F3 = 2.0;
}

function mul(block_token)
{
        this.F3 = eval(this.F5) * this.F0;
}

function sub1(block_token)
{
        this.F4 = this.F1 - this.F2;
}

function sub2(block_token)
{
        this.F5 = this.F3 - this.F6;
}

var pre1 = 1.0;
function int1(block_token)
{
        pre1 = this.F4*0.01 + pre1;
        this.F2 = pre1;
        this.F6 = pre1;
}

var pre2 = 1.0;
function int2(block_token)
{

        pre2 = this.F5*0.01 + pre2;
        this.F4 = pre2;
        out_F6 = pre2;
}
```

Figure B-4. User input JavaScript for the FBM in Figure B-1

```
<?xml version="1.0" encoding="UTF-8"?>
<DXL xsi:schemaLocation="'dxl.xsd"'>
  <block id="F0" type="SYNC" >
            <port id="F0.OP1" type="OUTPUT" target="F2.IP1" data_type="FLOAT"></port>
            <definition id="input.js" func="con1" start="TRUE"></definition> </block>
  <block id="F1" type="SYNC" >
            <port id="F1.OP1" type="OUTPUT" target="F3.IP1" data_type="FLOAT"></port>
            <definition id="input.js" func="con2" start="TRUE"></definition> </block>
  <block id="F2" type="SYNC" >
            <port id="F2.IP1" type="INPUT" source="F0.OP1" data_type="FLOAT"></port>
            <port id="F2.OP1" type="OUTPUT" target="F3.IP2" data_type="FLOAT"></port>
            <port id="F2.IP2" type="INPUT" source="F5.OP1" data_type="FLOAT" initial="1.0"></port>
            <definition id="input.js" func="mul"></definition></block>
  <block id="F3" type="SYNC" >
            <port id="F3.IP1" type="INPUT" source="F1.OP1" data_type="FLOAT"></port>
            <port id="F3.IP2" type="INPUT" source="F2.OP1" data_type="FLOAT"></port>
            <port id="F3.OP1" type="OUTPUT" target="F4.IP1" data_type="FLOAT"></port>
            <definition id="input.js" func="sub1"></definition></block>
  <block id="F4" type="SYNC" >
            <port id="F4.IP1" type="INPUT" source="F3.OP1" data_type="FLOAT"></port>
            <port id="F4.OP1" type="OUTPUT" target="F5.IP1" data_type="FLOAT"></port>
            <port id="F4.IP2" type="INPUT" source="F6.OP1" data_type="FLOAT" initial="1.0"></port>
            <definition id="input.js" func="sub2"></definition></block>
  <block id="F5" type="SYNC" >
            <port id="F5.IP1" type="INPUT" source="F4.OP1" data_type="FLOAT"></port>
            <port id="F5.OP1" type="OUTPUT" target="F2.IP2" data_type="FLOAT"></port>
            <port id="F5.OP2" type="OUTPUT" target="F6.IP1" data_type="FLOAT"></port>
            <definition id="input.js" func="int1"></definition></block>
  <block id="F6" type="SYNC" >
            <port id="F6.IP1" type="INPUT" source="F5.OP2" data_type="FLOAT"></port>
            <port id="F6.OP1" type="OUTPUT" target="F4.IP2" data_type="FLOAT"></port>
            <port id="F6.OP2" type="OUTPUT" target="EX.IP1" data_type="FLOAT"></port>
            <definition id="input.js" func="int2"></definition></block>
  <connect id="TR02">
            <port id="TR02.OP1" type="INPUT" source="F0.OP1" data_type="FLOAT"></port>
            <port id="TR02.IP1" type="OUTPUT" source="F2.IP1" data_type="FLOAT"></port> </connect>
  <connect id="TR13">
            <port id="TR13.OP1" type="INPUT" source="F1.OP1" data_type="FLOAT"></port>
            <port id="TR13.IP1" type="OUTPUT" source="F3.IP1" data_type="FLOAT"></port></connect>
  <connect id="TR23">
            <port id="TR23.OP1" type="INPUT" source="F2.OP1" data_type="FLOAT"></port>
            <port id="TR23.IP1" type="OUTPUT" source="F3.IP2" data_type="FLOAT"></port></connect>
  <connect id="TR34">
            <port id="TR34.OP1" type="INPUT" source="F3.OP1" data_type="FLOAT"></port>
            <port id="TR34.IP1" type="OUTPUT" source="F4.IP1" data_type="FLOAT"></port></connect>
  <connect id="TR45">
            <port id="TR45.OP1" type="INPUT" source="F4.OP1" data_type="FLOAT"></port>
            <port id="TR45.IP1" type="OUTPUT" source="F5.IP1" data_type="FLOAT"></port></connect>
  <connect id="TR52">
            <port id="TR52.OP1" type="INPUT" source="F5.OP1" data_type="FLOAT"></port>
            <port id="TR52.IP1" type="OUTPUT" source="F2.IP2" data_type="FLOAT"></port></connect>
  <connect id="TR56">
            <port id="TR56.OP1" type="INPUT" source="F5.OP2" data_type="FLOAT"></port>
            <port id="TR56.IP1" type="OUTPUT" source="F6.IP1" data_type="FLOAT"></port></connect>
  <connect id="TR64">
            <port id="TR64.OP1" type="INPUT" source="F6.OP1" data_type="FLOAT"></port>
            <port id="TR64.IP1" type="OUTPUT" source="F4.IP2" data_type="FLOAT"></port></connect>
  <connect id="TR6X">
            <port id="TR6X.OP1" type="INPUT" source="F6.OP2" data_type="FLOAT"></port>
            <port id="TR6X.IP1" type="OUTPUT" source="EX.IP1" data_type="FLOAT"></port></connect>
  <simulation start_time="0.0" end_time="20.0" delta_time="0.01" cycle_time="0.01"/>
</DXL>
```

Figure B-5. DXL for the FBM in Figure B-1

```
var pre_x = 1;
var pre_x1 = 1;
var pre_x2 = 0;

var d_x;
var d_x1;
var d_x2;
var d_y = 0.1;

var x_path = "M 87.9586 318.85 L 98 318.85 " ;
var x1_path = "M 91.1 441.364 L 97.8121 441.364 ";
var x2_path = "M 91.1 554.453 L 97.8121 554.453 ";


function user_code()
{
/*****************************************************************************
 *          Modify this function.                                            *
 *          Use simulate output parameters to change the attributes of your SVG objects.          *
 *****************************************************************************/

          x0.firstChild.data = out_F6
          x1.firstChild.data = out_F5;
          x2.firstChild.data = out_F4;

          F6.setAttribute('opacity',(((current_time*100)%10)/10)+0.3);
          F5.setAttribute('opacity',((((current_time+0.03)*100)%10)/10)+0.3);
          F4.setAttribute('opacity',((((current_time+0.06)*100)%10)/10)+0.3);
          F3.setAttribute('opacity',((((current_time)*100)%10)/10)+0.3);
          F2.setAttribute('opacity',((((current_time+0.03)*100)%10)/10)+0.3);
          F1.setAttribute('opacity',((((current_time+0.03)*100)%10)/10)+0.3);
          F0.setAttribute('opacity',((((current_time+0.06)*100)%10)/10)+0.3);

          if(current_time < 19.99)
          {
                    d_x = (pre_x - out_F6) * 51;
                    d_x1 = (pre_x1 - out_F5) * 40;
                    d_x2 = (pre_x2 - out_F4) * 26;

                    d_x = (d_x + '').substring(0,5);
                    d_x1 = (d_x1 + '').substring(0,5);
                    d_x2 = (d_x2 + '').substring(0,5);

                    x_path = x_path + ' l ' + d_y + ' ' + d_x;
                    x1_path = x1_path + ' l ' + d_y +  ' ' + d_x1;
                    x2_path = x2_path + ' l ' + d_y+ ' ' + d_x2;

                    xstart.setAttribute('d',x_path);
                    x1start.setAttribute('d',x1_path);
                    x2start.setAttribute('d',x2_path);

                    pre_x = out_F6;
                    pre_x1 = out_F5;
                    pre_x2 = out_F4;


          }
          //window.parent.plot(current_time,out_F6);
}
```

Figure B-6. User simulation code in merged SVG

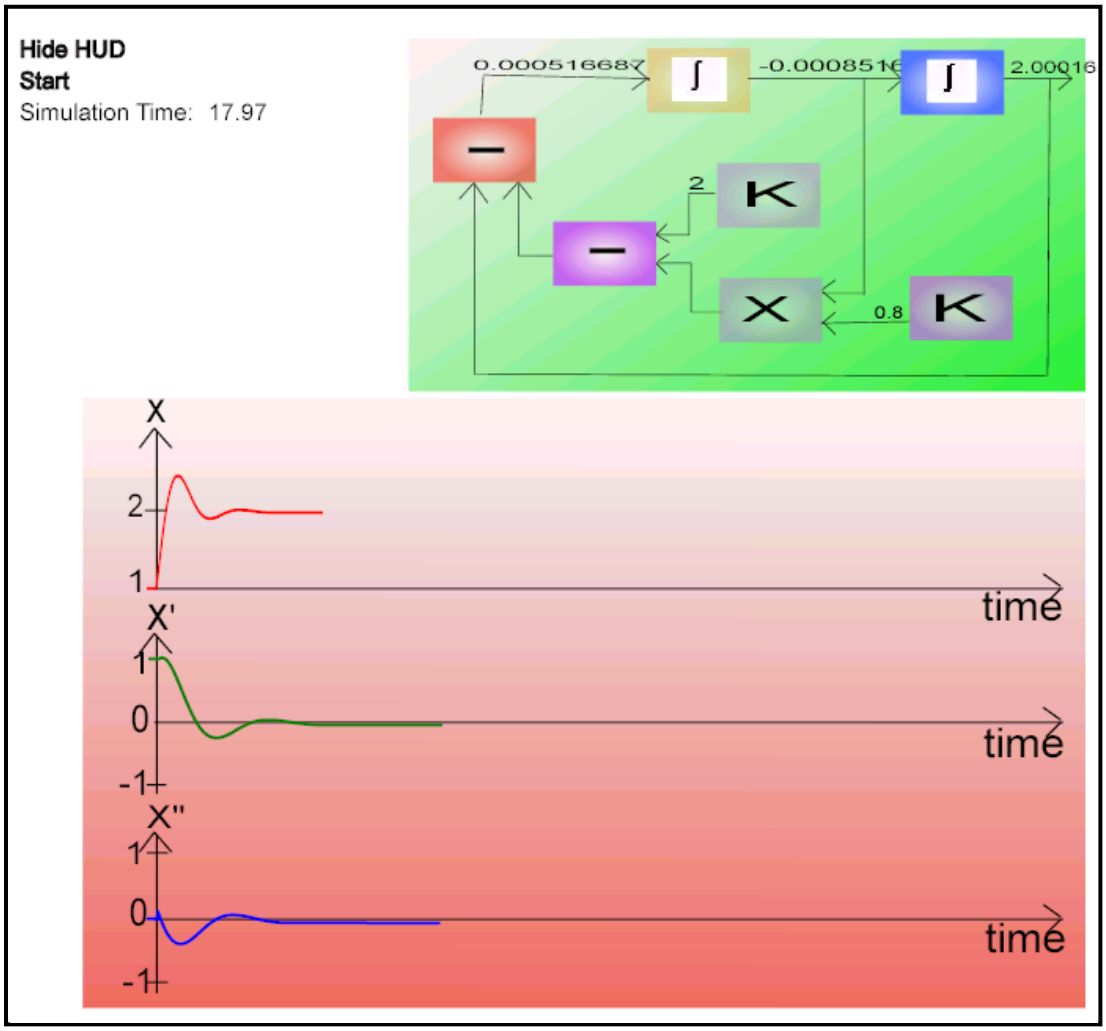

Figure B-7. Final simulation

Multi Model: Finite State Machine nested in Functional Block Model

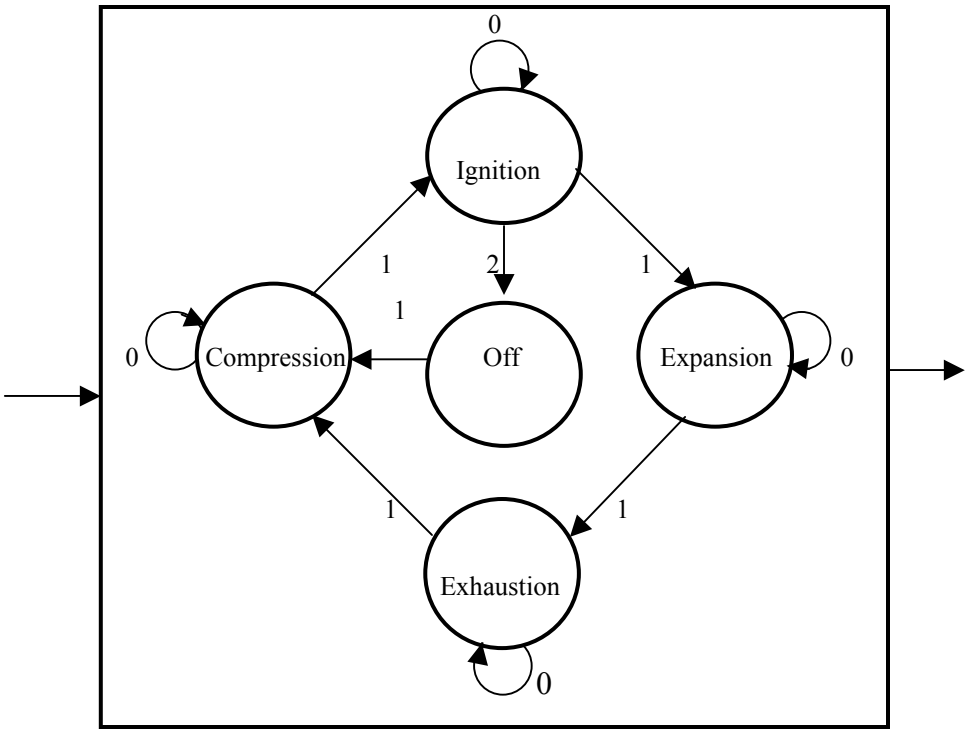

Figure B-8. Diagram for the four stroke gasoline or diesel engine

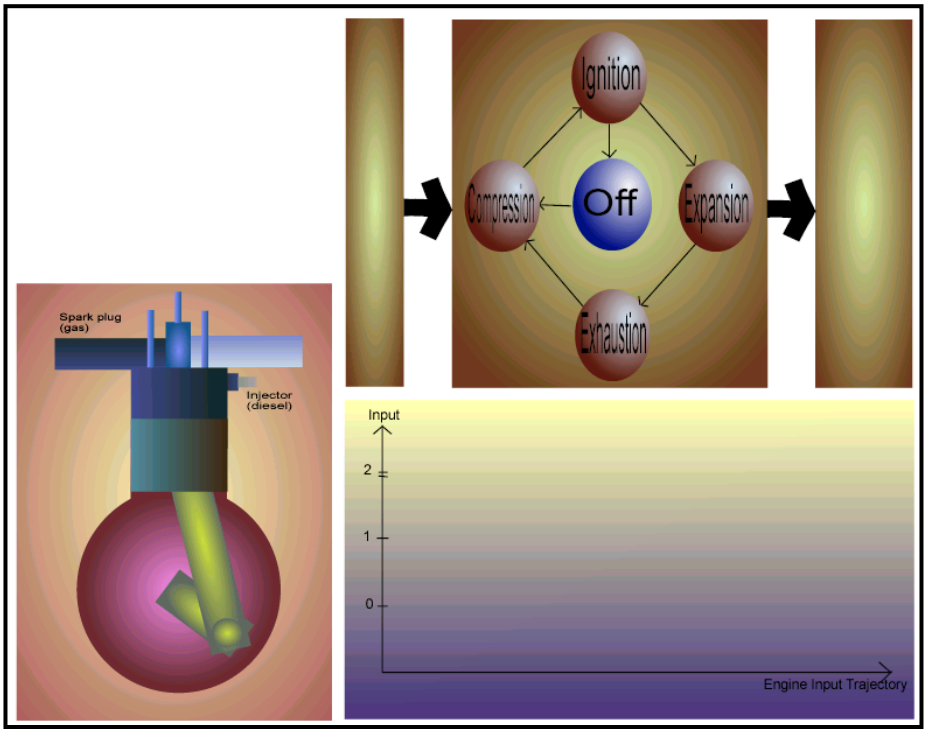

Figure B-9. SVG scene for the multimodel in Figure B-8

```
<?xml version="1.0" encoding="utf-8"?>
<MXL>
 <model type="FBM">
  <topology type="GRAPH">
   <node id="F1" type="FUNCTION"><script id="engine.js" func="gen"/></node>
   <node id="F2" type="FUNCTION">
    <model id="F2FSM" type="FSM">
     <topology type="GRAPH">
      <node id="Off" type="STATE"  data_type="BOOLEAN"  start="TRUE">
       <script id="engine.js" func="off"/>
      </node>
      <node id="Compression" type="STATE" data_type="BOOLEAN">
       <script id="engine.js" func="compression"/>
      </node>
      <node id="Ignition" type="STATE" data_type="BOOLEAN">
       <script id="engine.js" func="ignition"/>
      </node>
      <node id="Expansion" type="STATE" data_type="BOOLEAN">
       <script id="engine.js" func="expansion"/>
      </node>
      <node id="Exhaustion" type="STATE" data_type="BOOLEAN">
       <script id="engine.js" func="exhaustion"/>
      </node>
      <edge id="Off1" type="TRANSITION" begin="Off" end="Compression" data_type="BOOLEAN">
       <script id="engine.js" func="off2Compression"/>
      </edge>
      <edge id="Compression0" type="TRANSITION" begin="Compression" end="Compression" data_type="BOOLEAN">
       <script id="engine.js" func="compression2Compression"/>
      </edge>
      <edge id="Compression1" type="TRANSITION" begin="Compression" end="Ignition" data_type="BOOLEAN">
       <script id="engine.js" func="compression2Ignition"/>
      </edge>
      <edge id="Ignition0"  type="TRANSITION" begin="Ignition" end="Ignition" data_type="BOOLEAN">
       <script id="engine.js" func="ignition2Ignition"/>
      </edge>
      <edge id="Ignition1" type="TRANSITION" begin="Ignition" end="Expansion" data_type="BOOLEAN">
       <script id="engine.js" func="ignition2Expansion"/>
      </edge>
      <edge id="Ignition2"  type="TRANSITION" begin="Ignition" end="Off"  data_type="BOOLEAN">
       <script id="engine.js" func="ignition2Off"/>
      </edge>
      <edge id="Expansion0" type="TRANSITION" begin="Expansion" end="Expansion" data_type="BOOLEAN">
       <script id="engine.js" func="expansion2Expansion"/>
      </edge>
      <edge id="Expansion1"  type="TRANSITION" begin="Expansion" end="Exhaustion" data_type="BOOLEAN">
       <script id="engine.js" func="expansion2Exhaustion"/>
      </edge>
      <edge id="Exhaustion0" type="TRANSITION" begin="Exhaustion" end="Exhaustion" data_type="BOOLEAN">
       <script id="engine.js" func="exhaustion2Exhaustion"/>
      </edge>
      <edge id="Exhaustion1"   type="TRANSITION" begin="Exhaustion" end="Compression" data_type="BOOLEAN">
       <script id="engine.js" func="exhaustion2Compression"/>
      </edge>
     </topology>
    </model>
   </node>
   <node id="F3" type="FUNCTION"><script id="engine.js" func="disp"/></node>
   <edge id="TR12" type="TRACE" begin="F1" end="F2" data_type="INTEGER"/>
   <edge id="TR23" type="TRACE" begin="F2" end="F3" data_type="STRING"/>
   <edge id="TR3X" type="TRACE" begin="F3" end="EX" data_type="BOOLEAN"/>
  </topology>
 </model>
 <simulation  start_time = "0" end_time="151" delta_time="1" cycle_time="20"/>
</MXL>
```

Figure B-10. MXL for the multimodel in Figure B-8

```
var CS = 'Off';
function gen()
{
        if(CS == 'Ignition') this.F2 = Math.floor(Math.random()*3);
        else if(CS == 'Off') this.F2 = 1;
        else this.F2 = Math.floor(Math.random()*2);
}
function disp() {           his.EX_OUT = this.F2;                                   }
function off() {            out_F2_Off = true;    CS = 'Off';                       }
function compression(){     out_F2_Compression = true;    CS = 'Compression';       }
function ignition() {       out_F2_Ignition = true;       CS = 'Ignition';          }
function expansion() {      out_F2_Expansion = true;      CS = 'Expansion';         }
function exhaustion() {     out_F2_Exhaustion = true;     CS = 'Exhaustion';        }
function off2Compression()
{
        if((parseInt(this.INPUT) == 1))   this.OUT = true;
        else   this.OUT = false;
}
function compression2Compression()
{
        if(parseInt(this.INPUT) == 0)   this.OUT = true;
        else   this.OUT = false;
}
function compression2Ignition()
{
        if((parseInt(this.INPUT) == 1))   this.OUT = true;
        else   this.OUT = false;
}
function ignition2Ignition()
{
        if(parseInt(this.INPUT) == 0)   this.OUT = true;
        else   this.OUT = false;
}
function ignition2Expansion()
{
        if(parseInt(this.INPUT) == 1)   this.OUT = true;
        else   this.OUT = false;
}
function ignition2Off()
{
        if(parseInt(this.INPUT) == 2)   this.OUT = true;
        else   this.OUT = false;
}
function expansion2Expansion()
{
        if(parseInt(this.INPUT) == 0)   this.OUT = true;
        else   this.OUT = false;
}
function expansion2Exhaustion()
{
        if((parseInt(this.INPUT) == 1))   this.OUT = true;
        else   this.OUT = false;
}
function exhaustion2Exhaustion()
{
        if(parseInt(this.INPUT) == 0)   this.OUT = true;
        else   this.OUT = false;
}
function exhaustion2Compression()
{
        if((parseInt(this.INPUT) == 1))   this.OUT = true;
        else   this.OUT = false;
}
```

Figure B-11. User input JavaScript for the multimodel in Figure B-8

```
<?xml version="1.0" encoding="UTF-8"?>
<DXL xsi:schemaLocation="'dxl.xsd"'>
  <block id="F1" type="SYNC" >
            <port id="F1.OP1" type="OUTPUT" target="F2.IP1" data_type="INTEGER"></port>
            <definition id="engine.js" func="gen"></definition></block>
  <block id="F2" type="SYNC" >
            <port id="F2.IP1" type="INPUT" source="F1.OP1" target="INPUT_FEEDER.IP1" data_type="INTEGER"></port>
            <port id="F2.OP1" type="OUTPUT" target="F3.IP1" data_type="STRING"></port>
            <DXL id="F2FSM" >
                <block id="INPUT_FEEDER" type="SYNC" internal="YES">
                    <port id="INPUT_FEEDER.IP1" type="INPUT" source="F2.IP1"></port>
                    <port id="INPUT_FEEDER.OP1" type="OUTPUT" target="INPUT.IP1"></port></block>
                <block id="INPUT" type="SYNC" internal="YES">
                    <port id="INPUT.IP1" type="INPUT" source="INPUT_FEEDER.OP1"></port>
                    <port id="INPUT.IP2" type="INPUT" source="OUT.OP6" initial = "Off"></port>
                    <port id="INPUT.OP1" type="OUTPUT" target="Off1.IP1" data_type="BOOLEAN"></port>
                    <port id="INPUT.OP2" type="OUTPUT" target="Compression0.IP1" data_type="BOOLEAN"></port>
                    <port id="INPUT.OP2" type="OUTPUT" target="Compression1.IP1" data_type="BOOLEAN"></port>
                    <port id="INPUT.OP3" type="OUTPUT" target="Ignition0.IP1" data_type="BOOLEAN"></port>
                    <port id="INPUT.OP3" type="OUTPUT" target="Ignition1.IP1" data_type="BOOLEAN"></port>
                    <port id="INPUT.OP3" type="OUTPUT" target="Ignition2.IP1" data_type="BOOLEAN"></port>
                    <port id="INPUT.OP4" type="OUTPUT" target="Expansion0.IP1" data_type="BOOLEAN"></port>
                    <port id="INPUT.OP4" type="OUTPUT" target="Expansion1.IP1" data_type="BOOLEAN"></port>
                    <port id="INPUT.OP5" type="OUTPUT" target="Exhaustion0.IP1" data_type="BOOLEAN"></port>
                    <port id="INPUT.OP5" type="OUTPUT" target="Exhaustion1.IP1" data_type="BOOLEAN"></port></block>
                <block id="OUT" type="ASYNC" internal="YES">
                    <port id="OUT.IP2" type="INPUT" source="Off1.OP1" data_type="BOOLEAN"></port>
                    <port id="OUT.IP2" type="INPUT" source="Compression0.OP1" data_type="BOOLEAN"></port>
                    <port id="OUT.IP3" type="INPUT" source="Compression1.OP1" data_type="BOOLEAN"></port>
                    <port id="OUT.IP3" type="INPUT" source="Ignition0.OP1" data_type="BOOLEAN"></port>
                    <port id="OUT.IP4" type="INPUT" source="Ignition1.OP1" data_type="BOOLEAN"></port>
                    <port id="OUT.IP1" type="INPUT" source="Ignition2.OP1" data_type="BOOLEAN"></port>
                    <port id="OUT.IP4" type="INPUT" source="Expansion0.OP1" data_type="BOOLEAN"></port>
                    <port id="OUT.IP5" type="INPUT" source="Expansion1.OP1" data_type="BOOLEAN"></port>
                    <port id="OUT.IP5" type="INPUT" source="Exhaustion0.OP1" data_type="BOOLEAN"></port>
                    <port id="OUT.IP2" type="INPUT" source="Exhaustion1.OP1" data_type="BOOLEAN"></port>
                    <port id="OUT.OP1" type="OUTPUT" target="Off.IP1"></port>
                    <port id="OUT.OP2" type="OUTPUT" target="Compression.IP1"></port>
                    <port id="OUT.OP3" type="OUTPUT" target="Ignition.IP1"></port>
                    <port id="OUT.OP4" type="OUTPUT" target="Expansion.IP1"></port>
                    <port id="OUT.OP5" type="OUTPUT" target="Exhaustion.IP1"></port>
                    <port id="OUT.OP6" type="OUTPUT" target="INPUT.IP2"></port>
                    <port id="OUT.OP6" type="OUTPUT" target="F2.OP1"></port></block>
                <block id="Off" type="SYNC" >
                    <port id="Off.IP1" type="INPUT" source="OUT.OP1"></port>
                        <definition id="engine.js" func="off"></definition></block>
                <block id="Compression" type="SYNC" >
                    <port id="Compression.IP1" type="INPUT" source="OUT.OP2"></port>
                        <definition id="engine.js" func="compression"></definition></block>
                <block id="Ignition" type="SYNC" >
                    <port id="Ignition.IP1" type="INPUT" source="OUT.OP3"></port>
                        <definition id="engine.js" func="ignition"></definition></block>
                <block id="Expansion" type="SYNC" >
                    <port id="Expansion.IP1" type="INPUT" source="OUT.OP4"></port>
                        <definition id="engine.js" func="expansion"></definition></block>
                <block id="Exhaustion" type="SYNC" >
                    <port id="Exhaustion.IP1" type="INPUT" source="OUT.OP5"></port>
                        <definition id="engine.js" func="exhaustion"></definition></block>
                <block id="Off1" type="SYNC" >
                    <port id="Off1.IP1" type="INPUT" source="INPUT.OP1" data_type="BOOLEAN"></port>
                    <port id="Off1.OP1" type="OUTPUT" target="OUT.IP2" data_type="BOOLEAN"></port>
                        <definition id="engine.js" func="off2Compression"></definition></block>
                <block id="Compression0" type="SYNC" >
                    <port id="Compression0.IP1" type="INPUT" source="INPUT.OP2" data_type="BOOLEAN"></port>
                    <port id="Compression0.OP1" type="OUTPUT" target="OUT.IP2" data_type="BOOLEAN"></port>
                        <definition id="engine.js" func="compression2Compression"></definition></block>
```

Figure B-12. DXL for the multimodel in Figure B-8

```
            <block id="Compression1" type="SYNC" >
                <port id="Compression1.IP1" type="INPUT" source="INPUT.OP2" data_type="BOOLEAN"></port>
                <port id="Compression1.OP1" type="OUTPUT" target="OUT.IP3" data_type="BOOLEAN"></port>
                    <definition id="engine.js" func="compression2Ignition"></definition></block>
            <block id="Ignition0" type="SYNC" >
                <port id="Ignition0.IP1" type="INPUT" source="INPUT.OP3" data_type="BOOLEAN"></port>
                <port id="Ignition0.OP1" type="OUTPUT" target="OUT.IP3" data_type="BOOLEAN"></port>
                    <definition id="engine.js" func="ignition2Ignition"></definition></block>
            <block id="Ignition1" type="SYNC" >
                <port id="Ignition1.IP1" type="INPUT" source="INPUT.OP3" data_type="BOOLEAN"></port>
                <port id="Ignition1.OP1" type="OUTPUT" target="OUT.IP4" data_type="BOOLEAN"></port>
                    <definition id="engine.js" func="ignition2Expansion"></definition></block>
            <block id="Ignition2" type="SYNC" >
                <port id="Ignition2.IP1" type="INPUT" source="INPUT.OP3" data_type="BOOLEAN"></port>
                <port id="Ignition2.OP1" type="OUTPUT" target="OUT.IP1" data_type="BOOLEAN"></port>
                    <definition id="engine.js" func="ignition2Off"></definition></block>
            <block id="Expansion0" type="SYNC" >
                <port id="Expansion0.IP1" type="INPUT" source="INPUT.OP4" data_type="BOOLEAN"></port>
                <port id="Expansion0.OP1" type="OUTPUT" target="OUT.IP4" data_type="BOOLEAN"></port>
                    <definition id="engine.js" func="expansion2Expansion"></definition></block>
            <block id="Expansion1" type="SYNC" >
                <port id="Expansion1.IP1" type="INPUT" source="INPUT.OP4" data_type="BOOLEAN"></port>
                <port id="Expansion1.OP1" type="OUTPUT" target="OUT.IP5" data_type="BOOLEAN"></port>
                    <definition id="engine.js" func="expansion2Exhaustion"></definition></block>
            <block id="Exhaustion0" type="SYNC" >
                <port id="Exhaustion0.IP1" type="INPUT" source="INPUT.OP5" data_type="BOOLEAN"></port>
                <port id="Exhaustion0.OP1" type="OUTPUT" target="OUT.IP5" data_type="BOOLEAN"></port>
                    <definition id="engine.js" func="exhaustion2Exhaustion"></definition></block>
            <block id="Exhaustion1" type="SYNC" >
                <port id="Exhaustion1.IP1" type="INPUT" source="INPUT.OP5" data_type="BOOLEAN"></port>
                <port id="Exhaustion1.OP1" type="OUTPUT" target="OUT.IP2" data_type="BOOLEAN"></port>
                    <definition id="engine.js" func="exhaustion2Compression"></definition></block>
        </DXL>
</block>
<block id="F3" type="SYNC" >
        <port id="F3.IP1" type="INPUT" source="F2.OP1" data_type="STRING"></port>
        <port id="F3.OP1" type="OUTPUT" target="EX.IP1" data_type="BOOLEAN"></port>
        <definition id="engine.js" func="disp"></definition></block>
<connect id="TR12">
        <port id="TR12.OP1" type="INPUT" source="F1.OP1" data_type="INTEGER"></port>
        <port id="TR12.IP1" type="OUTPUT" source="F2.IP1" data_type="INTEGER"></port></connect>
<connect id="TR23">
        <port id="TR23.OP1" type="INPUT" source="F2.OP1" data_type="STRING"></port>
        <port id="TR23.IP1" type="OUTPUT" source="F3.IP1" data_type="STRING"></port></connect>
<connect id="TR3X">
        <port id="TR3X.OP1" type="INPUT" source="F3.OP1" data_type="BOOLEAN"></port>
        <port id="TR3X.IP1" type="OUTPUT" source="EX.IP1" data_type="BOOLEAN"></port></connect>
<simulation start_time="0" end_time="151" delta_time="1" cycle_time="20"/>
</DXL>
```

Figure B-12. Continued

```
var X;
var pathString;
var i = 1;
var preF1 = 2;
function user_code()
{
	/*********************************************************************************************
	 *	Modify this function.                                                                    *
	 *	Use simulate output parameters to change the attributes of your SVG objects.  *
	 *********************************************************************************************/

	if(out_F3 == 'Compression')
	{
		Off.setAttribute('fill','url(#radialGradient446)');
		Compression.setAttribute('fill','url(#radialGradient169)');
		Ignition.setAttribute('fill','url(#radialGradient446)');
		Expansion.setAttribute('fill','url(#radialGradient446)');
		Exhaustion.setAttribute('fill','url(#radialGradient446)');
		Rgas.setAttribute('fill','url(#linearGradient280)');
		Lgas.setAttribute('fill','url(#linearGradient280)');
		mover.setAttribute('y','330.496325');
		Gexpansion_s.setAttribute('transform',"matrix(0.794908,0.543846,-0.578367,0.747464,272.125,-25.5823)");
		expansion_center.setAttribute('x',"279.178931")
		expansion_center.setAttribute('y',"-462.431129");
		expansion_center.setAttribute('width',"34.553685");
		expansion_center.setAttribute('height',"79.316066");
		expansion_center.setAttribute('ry',"19.649390");
		expansion_center.setAttribute('transform',"matrix(-0.626842,0.779147,-0.779147,-0.626842,0,0)");
	}
	else if(out_F3 == 'Ignition')
	{

		Off.setAttribute('fill','url(#radialGradient446)');
		Compression.setAttribute('fill','url(#radialGradient446)');
		Ignition.setAttribute('fill','url(#radialGradient169)');
		Expansion.setAttribute('fill','url(#radialGradient446)');
		Exhaustion.setAttribute('fill','url(#radialGradient446)');
		Rgas.setAttribute('fill','url(#linearGradient280)');
		Lgas.setAttribute('fill','url(#linearGradient280)');
		mover.setAttribute('y','330');
		Gexpansion_s.setAttribute('transform',"matrix(0.987929,-0.107046,0.0472239,0.778967,-22.1638,87.8095)");
		expansion_center.setAttribute('x',"462.693992")
		expansion_center.setAttribute('y',"175.048865");
		expansion_center.setAttribute('width',"34.552925");
		expansion_center.setAttribute('height',"79.314325");
		expansion_center.setAttribute('ry',"19.648959");
		expansion_center.setAttribute('transform',"matrix(0.687992,0.725718,-0.725718,0.687992,0,0)");
	}
	else if(out_F3 == 'Exhaustion')
	{
		Off.setAttribute('fill','url(#radialGradient446)');
		Compression.setAttribute('fill','url(#radialGradient446)');
		Ignition.setAttribute('fill','url(#radialGradient446)');
		Expansion.setAttribute('fill','url(#radialGradient446)');
		Exhaustion.setAttribute('fill','url(#radialGradient169)');
		Rgas.setAttribute('fill','url(#linearGradient272)');
		Lgas.setAttribute('fill','url(#linearGradient280)');
		mover.setAttribute('y','356.496325');
		Gexpansion_s.setAttribute('transform',"matrix(0.888014,0.459802,-0.459802,0.888014,189.193,-26.3495)");
		expansion_center.setAttribute('x',"-506.572298")
		expansion_center.setAttribute('y',"-274.316330");
		expansion_center.setAttribute('width',"34.553702");
		expansion_center.setAttribute('height',"79.316094");
		expansion_center.setAttribute('ry',"19.649397");
		expansion_center.setAttribute('transform',"matrix(-0.650595,-0.759425,0.759425,-0.650595,0,0)");
	}
```

Figure B-13. User simulation code in merged SVG

```
	else if(out_F3 == 'Expansion')
	{
		Off.setAttribute('fill','url(#radialGradient446)');
		Compression.setAttribute('fill','url(#radialGradient446)');
		Ignition.setAttribute('fill','url(#radialGradient446)');
		Expansion.setAttribute('fill','url(#radialGradient169)');
		Exhaustion.setAttribute('fill','url(#radialGradient446)');
		Rgas.setAttribute('fill','url(#linearGradient280)');
		Lgas.setAttribute('fill','url(#linearGradient272)');
		mover.setAttribute('y','356.496325');
		Gexpansion_s.setAttribute('transform',"matrix(0.999771,0.0213429,-0.0213429,0.999771,13.1986,-
		0.517173)");
		expansion_center.setAttribute('x',"294.119613")
		expansion_center.setAttribute('y',"-497.355633");
		expansion_center.setAttribute('width',"34.553922");
		expansion_center.setAttribute('height',"79.316607");
		expansion_center.setAttribute('ry',"19.649525");
		expansion_center.setAttribute('transform',"matrix(-0.597556,0.801827,-0.801827,-0.597556,0,0)");
	}
	else if(out_F3 == 'Off')
	{
		Off.setAttribute('fill','url(#radialGradient169)');
		Compression.setAttribute('fill','url(#radialGradient446)');
		Ignition.setAttribute('fill','url(#radialGradient446)');
		Expansion.setAttribute('fill','url(#radialGradient446)');
		Exhaustion.setAttribute('fill','url(#radialGradient446)');
	}

	//FSM simulate

	input.firstChild.data = out_F1;
	output.firstChild.data = out_F3;

	//Engine Input Trajectory simulate
	if(current_time < 150)
	{
		if(i == 1)
		{
			X = "M 329.845 405.238 L 337.698 406.023";
			i = 0;
		}
		if(out_F1 == 0)
		{
			if(preF1 == 2){		X = X + " l 0 115 l 2.7 0";		}
			else if(preF1 == 1){	X = X + " l 0 57.5 l 2.7 0";		}
			else if(preF1 == 0){	X = X + " l 2.7 0";
			preF1 = 0;
		}
		else if(out_F1 == 1)
		{
			if(preF1 == 2){		X = X + " l 0 57.5 l 2.7 0";		}
			else if(preF1 == 1){	X = X + " l 2.7 0";			}
			else if(preF1 == 0){	X = X + " l 0 -57.5 l 2.7 0";		}
			preF1 = 1;
		}
		else if(out_F1 == 2)
		{
			if(preF1 == 2){		X = X + " l 2.7 0";			}
			else if(preF1 == 1){	X = X + " l 0 -57.5 l 2.7 0";		}
			else if(preF1 == 0){	X = X + " l 0 -115 l 2.7 0";		}
			preF1 = 2;
		}
		trajectory.setAttribute('d',X);
	}

}
```

Figure B-13. Continued

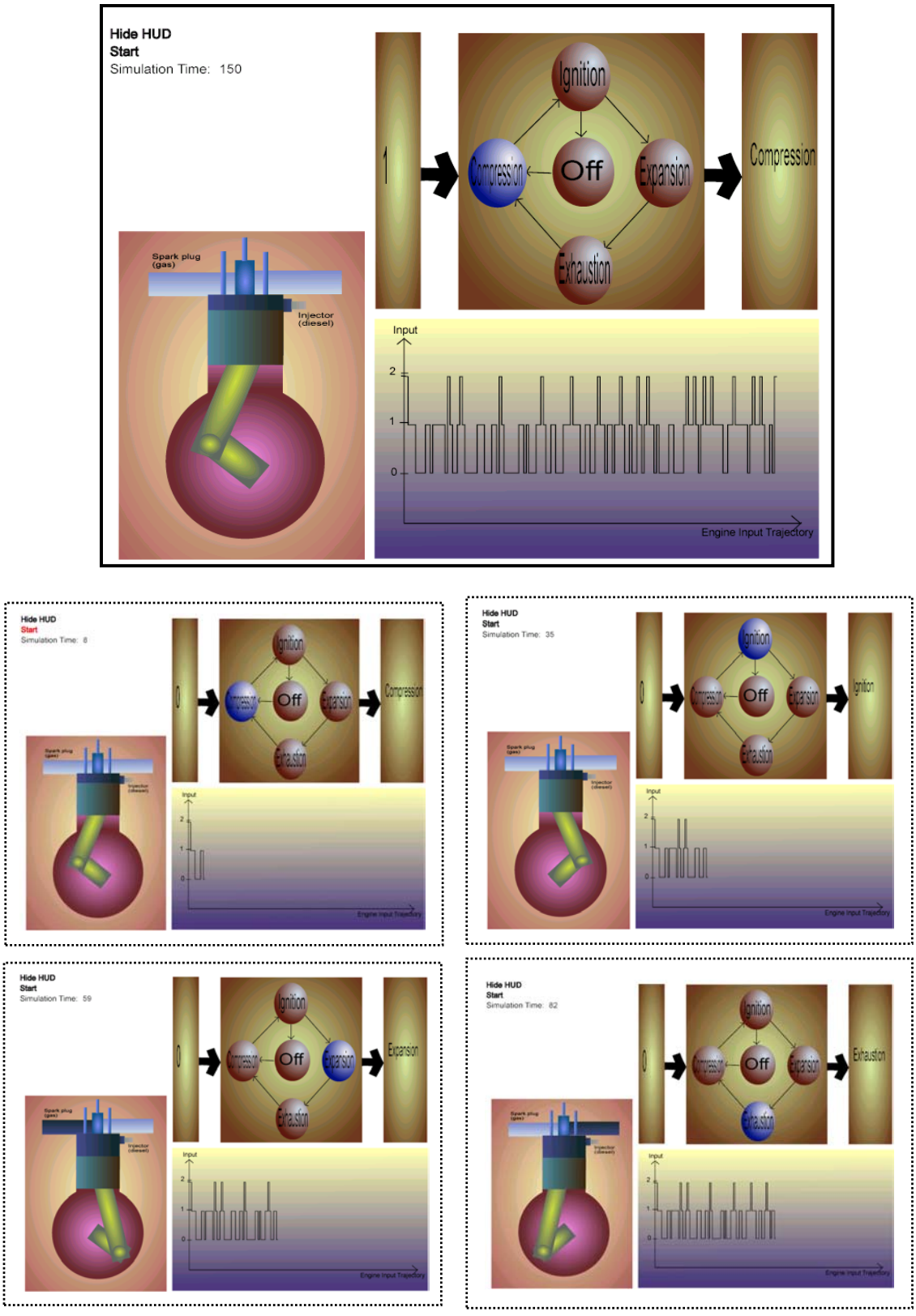

Figure B-14. Final simulation.

LIST OF REFERENCES

[1] World Wide Web Consortium (W3C). 2000. Extensible Markup Language (XML). Available from URL: http://www.w3.org/TR/REC-xml. Site last visited June 2003.

[2] World Wide Web Consortium (W3C). 1999. The Extensible Stylesheet Language (XSL). Available from URL: http://www.w3.org/Style/XSL. Site last visited June 2003.

[3] Kay, M. 2003. SAXSON: The XSLT Processor. Available from URL: http://saxon.sourceforge.net. Site last visited June 2003.

[4] The Apache Software Foundation. 2003. Xalan-Java. Available from URL: http://xml.apache.org/xalan-j/index.html. Site last visited June 2003.

[5] World Wide Web Consortium (W3C). 2001. Scalable Vector Graphics (SVG) 1.0 Secification. Available from URL: http://www.w3.org/TR/SVG/. Site last visited June 2003.

[6] Kim, T., Fishwick, P. A. A 3D XML-Based Modeling and Simulation Framework for Dynamic Models, Web3D 2002;103-109.

[7] Hopkins, J. F. An Investigation of the use of Metaphor in the *rube*™ Paradigm [dissertation]. Gainesville (FL): University of Florida; 2001.

[8] Fishwick, P. A. Hopkins, J. F. The *rube* Framework for Personalized 3-D Software Visualization, Software Visualization 2002; 368-380

[9] World Wide Web Consortium (W3C). 2002. Extensible 3D (X3D) Graphics Working Group. Available from URL: http://www.web3d.org/x3d.html. Site last visited June 2003

[10] Park, M., Fishwick, P. A. "SimPackJ/S: A Web-Oriented Toolkit for Discrete Event Simulation", in Enabling Technologies for Simulation Science. SPIE 2002.

[11] Domokos, P., Varro, D. 2002. An Open Visualization Framework for Metamodel-Based Modeling languages. Electronic Notes in Theoretical Computer Science 72 (2). Available from URL: http://www.elsevier.nl/gej-ng/31/29/23/127/46/28/72.2.009.pdf. Site last visited June 2003

[12] Blais, C., Brutzman, D. 2001. Web-based 3D technology for scenario authoring and visualization: The SAVAGE Project. Proceedings of the 2001 Interservice/Industry Training, Simulation, and Education Conference (I/ITSEC). Available from URL: http://www.npsnet.org/Publications/Web3DSavage.DOC.pdf. Site last visited June 2003

[13] Balci, O. Dynamic Object Decomposition In The Visual Simulation Environment, Proceedings of the 11th European Simulation Multiconference1997; 69-73

[14] Shin, S. J., Lemon, O. 2002. Diagrams. The Stanford Encyclopedia of Philosophy. Available from URL: http://plato.stanford.edu/entries/diagrams. Site last visited June 2003

[15] Kulpa, Z. Diagrammatic representation and reasoning, Machine GRAPHICS & VISION 1994; 3: 77-103

[16] Kaplinski, L. 2003. Sodipodi. Available from URL: http://sodipodi.sourceforge.net. Site last visited June 2003

[17] Weiss, A. "Open Source Moves To The Mainstream: Reliability, performance, and availability draw developers away from commercial products", Information week 2000; April 10.

[18] Hopkins, J. F., Fishwick, P. A. A Three-Dimensional Synthetic Human Agent Metaphor for Modeling and Simulation. Proceedings of the IEEE 2001; 89(2): 131-147

[19] Fishwick, P. A. Aesthetic Computing: Making Artistic Mathematics and Software, YLEM Newsletter 2002; 22(10):6-11.

[20] Fishwick, P. A. Simulation Model Design and Execution: Building Digital Worlds. Englewood Cliffs: Prentice-Hall, 1995.

[21] Boyd, N.2000. Rhino: JavaScript for Java. Available from URL: http://www.mozilla.org/rhino. Site last visited June 2003

BIOGRAPHICAL SKETCH

Hyunju Shim was born on June 6th, 1977, in Kumsan, Republic of Korea. She received her Bachelor of Science degree in computer science from Sogang University, Seoul, Republic of Korea, in February 2001. In 2003, she received her Master of Science degree from the Department of Computer and Information Science and Engineering at the University of Florida. Her major research area is computer modeling and simulation.