RUBE_QM: A 3D SIMULATION AND MODELING APPROACH FOR QUEUING SYSTEMS

By

NAMKYU LIM

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

2004

To my country,

my mother, my wife, my daughter, my son,

my father-in-law, and my mother-in-law

ACKNOWLEDGMENTS

I would like to offer my sincere appreciation to my advisor, Dr. Paul A. Fishwick, for his insightful advice and suggestions in this research and for giving me the opportunity to work on this interesting topic. I express my deepest thanks to Dr. Joachim Hammer and Dr. Sumi Helal for serving on my committee.

I thank my Modeling and Simulation Lab colleagues, Minho Park, Jinho Lee, and Hyunju Shim, for their time and technical assistance. I would especially like to thank Jinho Lee for his continual advice during my research. I will never forget all of our intensive discussions.

I am most grateful to my country, my mother, my wife, and my father-in-law and my mother-in-law for their support, encouragement, and prayers.

TABLE OF CONTENTS

page

LIST OF FIGURES

Figure page

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

RUBE_QM: A 3D SIMULATION AND MODELING APPROACH
FOR QUEUING SYSTEMS

By

Namkyu Lim

August 2004

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

The RUBE architecture was developed in the Graphic, Modeling and Art Lab at the University of Florida to support a Web-based customized multimodel modeling and simulation development. An emergence of eXtensible Markup Language (XML) brought a new way of representing information and knowledge on the World Wide Web. XML is convenient for modeling and configuring graphical elements, and it opens a new method of sharing these entities over the Web. The RUBE architecture takes advantage of these XML technologies.

The goal of this research is to expand the supporting model type of RUBE to a queuing model. Currently, RUBE supports modeling and simulation of FBM (Functional Block Model) and FSM (Finite State Machine) model types with a 2D or 3D model representation. Since RUBE was built based on the XML technologies, it is easy to add new modeling and simulation development frameworks in the RUBE architecture. In order to add new model frameworks to the RUBE architecture, the architecture must be

compatible with the RUBE architecture. It must support the main goal of RUBE, which supports an easy construction of a personalized and customized model with a Web-based visualization.

To achieve the above requirement, RUBE_QM follows the methodology of the RUBE system. In order to support the queuing model, RUBE_QM needs to define the new model elements and semantics in Multimodel eXchange Language (MXL) written in XML. The XML document handling mechanism is used along with the name correspondence. Therefore MXL easily expands its model type definition.

In addition to the MXL definition, RUBE_QM requires a new data structure to store events for a given amount of time inside of the simulation environment. The RUBE_QM translator generates JavaScript files to support the queuing model event handling mechanism, and uses an event scheduling method to execute the model. Other model types in RUBE architecture are also executed by the event scheduling method. Since events in RUBE_QM are executed by the identical execution method as the other model types in RUBE, RUBE_QM can be used for modeling and simulation with heterogeneous model types.

The RUBE_QM is an expanded version of RUBE supporting FBM, FSM, and the queuing system modeling and simulation. The XML-based technologies are used with ease as leverage in expanding the application boundary of RUBE

CHAPTER 1
INTRODUCTION

1.1 Motivation

RUBE is a modeling and simulation framework for 2D or 3D dynamic models. The main purpose of RUBE is to provide users the freedom of presentation from the complicated modeling and simulation development. The RUBE project started in 2000 as a continuation of the research in multimodeling and Web–based modeling and simulation. To support multimodeling and Web-based modeling and simulation, RUBE initially employed Virtual Reality Modeling Language (VRML) for the model presentation and programming language. A special node of VRML, PROTO, was used to add behaviors of objects using ECMAScript (commonly known as JavaScript) [1]. The ROUTE function, which is provided by VRML, was used to connect the model behaviors to the model objects. To provide more expandability, RUBE evolved to an XML technology-based method. By the nature of XML, which supports separation of content from representation, RUBE easily supports decoupling of model definition from model presentation.

Up to this point, RUBE supports modeling and simulation of synchronous model types, such as FBM (Functional Block Model) and FSM (Finite State Machine). By adding the modeling and simulation frameworks for the queuing model to RUBE, RUBE expands its application boundary to the asynchronous model types. Adding a new modeling and simulation architecture to RUBE, while maintaining the current capability

of RUBE, is easily accomplished due to the fact that RUBE was based on the XML technologies.

The primary motivation of this research is to expand the application area of RUBE to the queuing model. The procedures to accomplish this goal are as follows. First, extract the queuing model components and their characteristics to define model semantics from their application fields. Second, based on the defined model semantics, find the behavioral elements, which control themselves or other model components. Third, make modeling and simulation frameworks. The most important thing while adding the new framework is that the new framework should be compatible with RUBE.

1.2 Organization of This Thesis

In Chapter 1, we give the motivation behind the thesis. We discuss the background knowledge of this thesis in Chapter 2. We introduce the related works in modeling and simulation in Chapter 3. In Chapter 4, we give the detailed discussion of RUBE_QM . In Chapter 5, we provide descriptions of the RUBE_QM model topology and the design of the model behavior. In Chapter 6, we discuss the modeling and simulation process in RUBE_QM. In Chapter 7, we offer our conclusions

CHAPTER 2
BACKGROUND KNOWLEDGE

2.1 Queuing Model (QM)

The queuing model is described as functional modeling using a function-based approach. The functional modeling relies on the functional elements as the building blocks upon which to construct a dynamic model [1]. *Queue*, *Customer*, and *Server* were described as the main elements of the queuing model by Randolph W.Hall in the queuing method [2]. The term *Customer* can refer to people or objects waiting for service, such as parts, machines, and aircraft. *Servers* are bank tellers, machine operators, maintenance mechanics or anyone else providing the service. Finally, *Queue* is a line of entities or a group of people waiting to be served [3]. To define the queuing system, the timing of the customer's arrival at the queue is defined as the arrival pattern. To define arrival patterns (interarrival time at queue between customers) of customers on the queue, random functions are usually used at the modeling and simulation. The server process defines the time taken to serve customers, commonly referred to as the service time. The standard notation for the queuing system is one involving a five-character code such as A/B/C/D/E: A defines arrival patterns, which means the interarrival time distribution; B defines the server process, which is the service-time distribution; C means the number of parallel servers; D represents the system capacity; and E refers the size of population. For example, the code M/M/1/∞/∞ defines a single server (1) system with unlimited queue capacity (∞) and the infinite queue size (∞). Interarrival times and service times are exponentially distributed (M). Using this notation, several types of queuing systems can

be defined: M/M/1, M/G/1, M/D/1, M/M/1/M, and M/M/c. The characters, defining service time and arrival pattern mean as follows: M indicates an exponential random distribution, G refers the general distributions, and D indicates a constant service time [4].

2.2 XML-based Technologies

2.2.1 eXtensible Markup Language (XML)

eXtensible Markup Language (XML) is a simple and flexible text format derived from Standard Generalized Markup language (SGML), originally designed to meet the challenges of large-scale electronic publishing. XML is playing an increasingly important role in the exchange of a wide variety of data on the Web [5].

The main goal of XML is to build upon the flexibility of SGML without the complexity and the presentation limitation of HyperText Markup Language (HTML). While HTML mainly represents information, XML deals with the nature of the information. This characteristic of XML allows users to manipulate the data in an XML document for the representation, distribution, and so forth [6]. To facilitate flexibility and transformability, XML provides a method to define its own tags and structure of documents using an XML *schema* [7]. It also provides a way to convert one XML document into a different XML document using an XSLT stylesheet [3]. The XML features, as previously described, motivate many research groups and enterprises to use XML as a tool or a software environment. RUBE uses XML as a core language to facilitate the separation of data from its presentation and to support transformability for modeling and simulation [6].

2.2.2 XML Schema

The XML *schema* is a structured definition for describing the structure of information. The definition can be described as an arrangement of tags and text in a document and applied to an entire class of documents in XML. The *schema* might also be viewed as an agreement of on a common vocabulary for a particular application that involves exchanging documents [7].

The structures of the XML document are described in terms of *constraints* in a *schema*. The *constraint* defines what can appear in any given context. Two kinds of *constraints* are used in a *schema* definition: *content definition* and *data type constraints*. *Content definition constraints* describe the order and the sequence of elements while *data type constraints* describe valid units of the data [8]. One of the purposes of the *schema* is to allow the machine validation of a document structure. A valid document means that it is suitable within the described definition of a class of documents. Any specific, individual document, which does not violate any of the *constraints* of the definition, is valid according to that *schema*. When applications are designed to send and receive much information via the Web, the ability of testing the validity of a document is an important aspect.

Also, the XML *schema* uses the XML elements and attributes to express the semantics of it. It can be edited and processed with the same tools that users employ to manipulate other XML documents.

2.2.3 eXtensible Stylesheet Language Transformation (XSLT)

eXtensible Stylesheet Language Transformations (XSLT) is a language which is primarily designed for transforming one XML document and its structure into another [9]. With XML, it is easy to create, read, and write to a document, but an intermediate

process is needed to convert one XML document into another type of document. To convert an XML document to the required document or structure, XSLT takes two steps: 1) the data are converted from the structure of the incoming XML document into a structure that reflects the desired structure; and 2) the new structure is used to generate an output in the required format such as JavaScript or PDF [10]. XSLT allows the data to be supplied to one of the new sources of applications that accept XML at the second step. XSLT relies on the XML parser to convert the XML document into a tree structure in which XSLT manipulates the document representation. XSLT translates an XML document into another document based on template rules that describe how each element must be processed for transformation. The rule for transformation expressed in XSLT is called a stylesheet.

The principal role of an XSLT processor is to apply an XSLT stylesheet to an XML source document and produce a result document with tree structure. The XSLT processor handles three types of trees: the source tree, the stylesheet tree, and the result tree [10]. RUBE uses xalan as an XSLT processor to support the transformation [11].

RUBE employs the following XML-based technologies: 1) a *schema* to define the model types in MXL (Multimodel eXchange Language), namely, MXL.xsd; and 2) XSLT to define the model translation template rule, MXL2DXL.xsl and MFE.xsl. The RUBE_QM framework also maintains the RUBE methodology to be compatible with other model types in RUBE and to support multimodel modeling and simulation. These topics will be discussed in detail in Chapters 3 and 4.

2.3 RUBE

The contributions of RUBE are providing a customized modeling and simulation tool and employing an XML-based specification for dynamic models [12]. The initial

RUBE used the VRML specific node and structure to facilitate dynamic multimodel constructions and implementation within the 3D immersive environment. RUBE changed the vehicle to XML to expand its application area. The architecture of RUBE is shown in Figure 2-1.

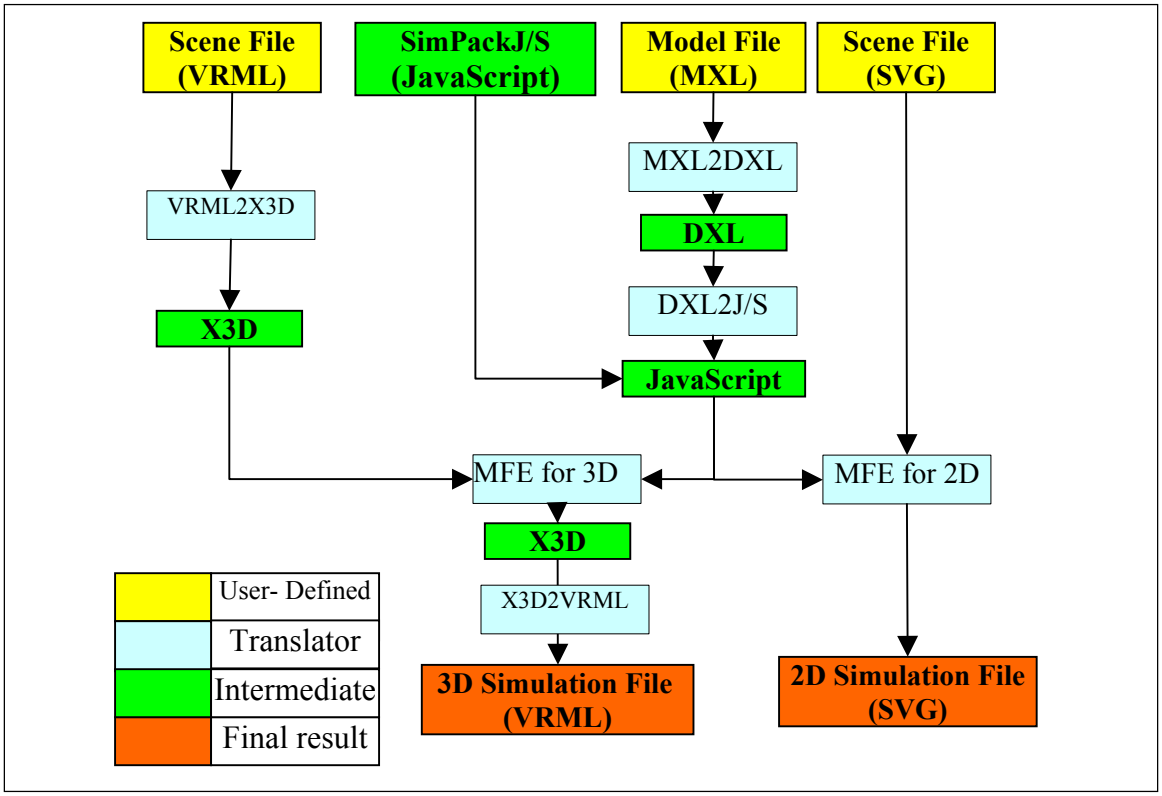

Figure 2-1. RUBE architecture

The modeling and simulation in RUBE consists two separate files: 1) a scene file, which defines model representation in a 2D or 3D modeling language; and 2) model files, which define the model semantics in Multimodel eXchange Language (MXL) [12, 13].

RUBE supports a 2D or 3D model representation. To create a scene file, users may use a 2D or 3D representation language such as Virtual Reality Modeling Language (VRML) or SVG (Scale Vector Graphic). VRML and SVC support the internal scripting in JavaScript and Java so that the scene can be presented dynamically using script. RUBE translates VRML into X3D by the translation step when the scene file is 3D geometry

language. When users define 2D scene file, the 2D scene file is moved directly into the Model Fusion Engine for 2D geometry (MFE for 2D) and transformed into a dynamic embedded 2D simulation model file. X3D is the XML version of VRML that describes the 3D geometry [13]. MXL defines the model topology and the dynamics of the model in the XML *schema*. Multimodeling in RUBE can be defined as a hierarchically connected set of dynamic behavioral models where each model is a specific type. The set of models are either homogeneous or heterogeneous [14]. In RUBE, when users define model file(s) and a scene file, the model file is translated into a homogeneous block diagram, namely, Dynamic eXchange Language (DXL) [12]. When the model file is translated into DXL, RUBE uses an XSLT stylesheet that defines the transformation rules of each model (MXL2DXL.xsl). DXL is the intermediate representation of model types between the user-defined model file and the RUBE-created simulation model in JavaScript. DXL is translated into a JavaScript simulation model using DXL.class in Java programming language. Document Object Model (DOM) is employed to generate the DXL.class file. JavaScript and the X3D or SVG file are fused with MFE.xsl to generate the script containing X3D files [12].

RUBE allows users to create the scene file in VRML or SVG. To visualize the X3D contents describing the 3D model geometry, RUBE uses a mechanism to convert the X3D file into the VRML file, which can be viewed in most Web-based browsers. By using the RUBE output, VRML scene objects can be controlled dynamically and simulated when users makes a connection. Users can make a connection between the model behavior JavaScript output and the model geometries using the VRML structure, such as ROUTE and PROTO. The ROUTE is a specific structure of VRML and can be

described as a pipe which links the model behavior output with a model scene or between model scenes [14].

CHAPTER 3
RELATED WORK

In this chapter, we discuss related topics in the queuing model modeling or XML-based modeling and simulation development toolkits. The Tool for Object–oriented Modeling And Simulation (TOMAS) is the process-oriented modeling and simulation development software which contains the queuing model element. The AToM^3 is a simulation and modeling software to support meta-modeling and multi-transformation. XML is used for transforming to other types of simulation languages. The Simulation Reference Simulator is a simulation execution environment to perform the simulation in SRML (Simulation Reference Markup Language). SRML uses XML to define simulation and modeling at Boeing. JavaSim will be discussed in the final section.

3.1 TOMAS: Tool for Object–oriented Modeling And Simulation

TOMAS is presented as a software package specially developed for the discrete event simulation of complex control problems in a logistics and a production environment. The model in TOMAS is described by means of a process-oriented approach and supports 3D animation in Delphi. The TOMAS provides *TomasElements* to define simulation elements using Delphi, and specific elements can be defined as descendents of a *TomasElement.* Each element contains implicit methods to control the sequencing mechanism [15].

The *TomasQueue* is a basic class to represent a set of *TomasElements* for modeling waiting lines, jobs in progress, and so forth. The *TomasQueue* automatically gather

statistics, such as the number in queue (minimum, mean, maximum) and waiting times (mean, maximum) for the model [16].

RUBE_QM is distinguished from TOMAS in terms of the methodology of modeling and simulation. RUBE_QM separates model semantics from presentation, and this gives the freedom to users to choose their own metaphor for the model. An aesthetic aspect of modeling can also be integrated with the model creation. However, TOMAS provides animation for verification and presentation with its own visual model.

3.2 A Meta-Modeling and Model-Transforming (AToM3)

The two main tasks of AToM^3 are *meta-modeling* and *model-transforming* proposed by Hans Vangheluwe and Juande Lara to support the distributed simulation and modeling. *Meta-modeling* refers to the description, or modeling of different kinds of formalisms used in modeling. *Model-transforming* refers to the (automatic) process of converting, translating, or modifying a model given in certain formalism into another model that might or might not be in the same formalism [17].

In AToM^3, formalisms and models are described as graphs. Relationships between model formalisms are described using Entities Relationship meta-model formalism (E-R). Constraint conditions are inserted in the E-R when constraints are needed. AToM^3 provides the graphical model definition and manipulation.

Model transformations are performed by graph rewriting, such as add Entities Relationship or constraint conditions. Individual model definitions (even E-R and constraints conditions) are saved and shared in the XML format while the transformation is performed in AToM^3 [18].

RUBE defines the relationship between models using model topology and XML-based schema whereas in AToM^3, the model E-R is defined in its own model formalism.

The main difference is that RUBE supports separation of presentation and model definition whereas AToM^3 supports distributed modeling and simulation even though each of them also uses XML technology and supports multimodeling.

3.3 Simulation Reference Simulator

Simulation Reference Markup Language (SRML) is an XML application that can describe the behavior for distributed simulation models. SRML uses XML and scripts to define the structure and behavior of all items comprising a simulation, such as items classes, event classes, scripts, and other elements for connecting and identifying behavioral elements [19]. SRML can be executed in a Simulation Reference Simulator (SR Simulator), which provides a runtime environment that includes the standardized XML Document Object Model (DOM). The runtime environment provides the item and event management, random number generation, mathematics and statistics, supporting plug-in, and so forth. The runtime environment intrinsically uses the DOM in item management and plug-in compilers for behavior.

Using the SR Simulator, a user can choose any host environment to create a simulation object and use its interface, such as Microsoft Internet Explorer, Microsoft Office applications, Microsoft Visual Basic, and Java [20].

SR Simulator supports defining model topology using its own tag. Users create a simulation program to control each item using encapsulated simulation objects control functions. SRML provides flexibility by allowing model developers to define their own XML schema rather than being constrained to use a specific set of tag names. As with SRML, RUBE gives the users the flexibility in defining models and the dynamics of the model, and it provides an executable Web-based simulation program and environment.

3.4 JavaSim

L.B. Arief and N.A. Speirs proposed a UML tool for an automatic generation of simulation programs. They developed a simulation framework called Simulation Modeling Language (SimML) to bridge the transformation from the design to the simulation programs. A UML tool that supports this framework has been constructed in the Java programming language using the JFC/Swing package [21], namely, JavaSim [22]. The simulation programs are generated in the Java programming language using JavaSim. JavaSim framework uses XML-based DTD to store the information and structure of the data. JavaSim also manipulates data using Java packages, such as the Simple API for XML (SAX) through IBM's XML4J parser. In this framework, new simulation objects and activities can be added to the components of SimML [23].

The JavaSim framework is constructed in UML, Java, and XML to use component information while RUBE is constructed mainly in an XML-based environment. RUBE supports dynamic visual modeling using the XML nature, which separates presentation and information from data. RUBE supports separation of model representation and model semantics, which gives the user the flexibility of model presentation.

CHAPTER 4
RUBE_QM FRAMEWORKS

In this chapter, the RUBE queuing model (RUBE_QM) frameworks are discussed in detail. In the first section, we introduce the description of RUBE_QM, and then we will discuss The RUBE_QM modeling components and the overall structure of the RUBE_QM frameworks in section 2 and 3, respectively.

4.1 RUBE_QM Overview

RUBE is an XML-based multimodel modeling and simulation development toolkit with a 2D or 3D visualization. RUBE supports multimodel modeling and simulation of the Functional Block Model (FBM) and Finite State Machine (FSM) with Web-based capability.

RUBE_QM adds the capability of modeling and simulation of the queuing model to RUBE. By adding the modeling and simulation frameworks for queuing model to RUBE, RUBE expands its application boundary to the asynchronous model types

The main development base of RUBE is XML. In RUBE, model representation files and model semantic files are defined in XML. New model semantics can be added with ease in the original RUBE model semantics. This comes from the XML document handling mechanism, name correspondence, which allows RUBE to work with different XML documents to copy and manipulate the corresponding elements from other XML documents [6].

RUBE_QM also supports a Web-based modeling and simulation as RUBE. The RUBE_QM structures can be divided into the model semantics and the representation as

in the original RUBE. The model semantics in the RUBE_QM architecture include queuing model semantics with the semantics of FBM and FSM model types.

4.2 RUBE_QM Model Elements

RUBE_QM extends the application area to events that have various time intervals and delays between event execution sequences. In order to support customized modeling and simulation of the queuing model, we need to ascertain what kinds of elements are needed to create the queuing model semantics. The general elements for the queuing model are shown in Figure 4-1.

Element	Usage	Attribute
<source>	Generates a stream of arriving customers, jobs, or entities.	<id> <Interarrivaltime>
<facility>	A system resource that acts as a collection of a queue and servers. It can contain one queue and one server or more.	<id>
<queue>	Data Structure for entities to wait until service is provided when demand for server exceeds the capacity of provided service.	<id> <Size>
<server>	A mechanism to provide services or processes to the arriving entities.	<id> <serviceTime>
<decision>	Make a decision to send entities (job, customer) positions using function for distribution.	<id>
<fork>	Pick a single fork output using a cumulative distribution generated from the probability attached to each branch using random number generation.	<id>
<join>	Collect specific number of entities and pass an entity or entities to the next step. This element can be used as a synchronizing function.	<id> <sizeOfCollection>
<sink>	Destroy event.	<id>

Figure 4-1. Model elements to define the queuing model semantics

I use the bank (computer system) for the example of the queuing model. When guests enter the bank, The guests (entities) may enter the bank (system) without any information (at random time interarrival) about how many customers are waiting for the service and

when the last guests (entities) entered in the bank. RUBE_QM needs to provide a function which generates interarrival time according to the user definition, which can be a specific time or a time generated by a random number generation function. This function can be defined using the **source** element. To define interarrival time between guests or entities, the **source** element needs an attribute such as **Interarrivaltime**. The guest must wait his turn to be served by the teller (server) at the designated waiting place or data structure. The waiting place or data structure is defined as a **queue** and teller (server) as a **server**. In some cases, the waiting line has an upper bound (finite queue size) to limit the waiting line. In order to support this case, the **Size** attribute is added to the queue element. Without any specific definition about the queue size, RUBE_QM acknowledges the **queue** size as infinite. To move into the **server**, the guest may need a service time with/without any relationship with the previous guest or entity. In this case, the **server** needs an attribute to define the service time function, which generates the service time as user-defined. This attribute name is **serviceTime**.

The **facility** is a collection of one **queue** and **server(s),** which acts only as a container of the queue and server(s). A **facility** has only one **queue** regardless of the number of **server**(s). The guest (entity) may want to be re-serviced or to proceed to the next service so RUBE_QM needs an element to tell the guest (entity) where to move. We define this action as the **decision** element.

The **fork** element can be used to move guests (entities) with the given probability. To pick a single fork output, a random number generation function can be used which is provided in SimPackJ/S [24].

In order to sort guests (entities) according to the entering sequence of the bank or system, the **join** element can be used. The guests (entities) are sorted by the non-decreasing order of the bank-entered time at **join** element. The **sink** element may be used to destroy the node after the entire event has been accomplished. The node types described earlier are defined using the XML schema with its required attributes. Figure 4-2 shows the model topology of RUBE_QM.

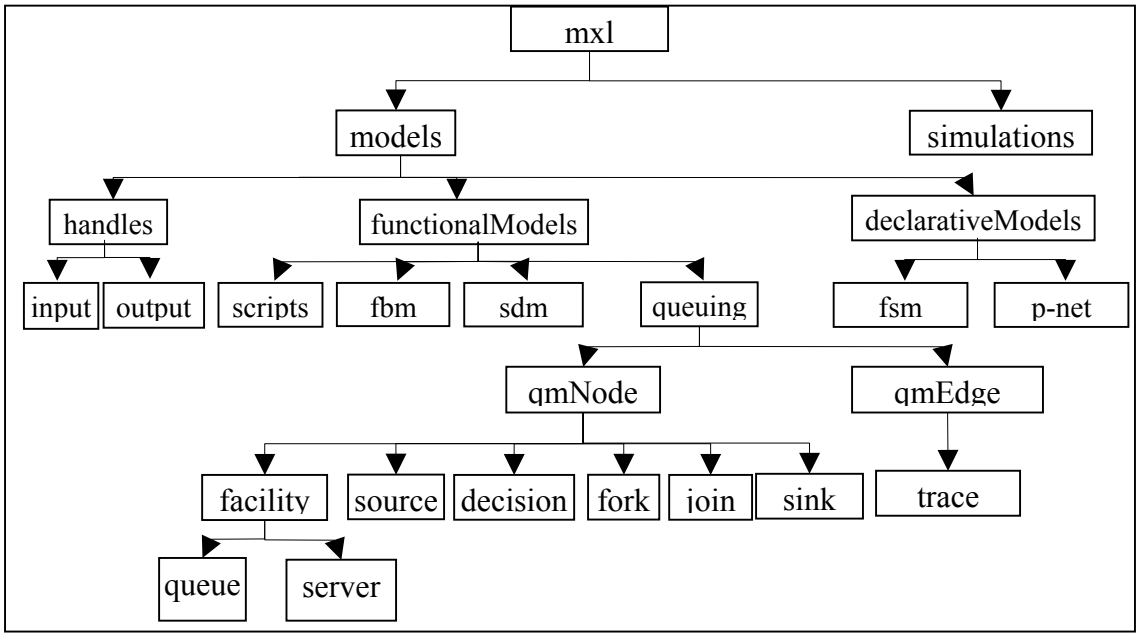

Figure 4-2. MXL topology of RUBE_QM

RUBE supports the modeling and simulation of FBM in the functional model category and FSM in the declarative model category. The queuing model is in the functional model category. With this model topology and attributes, the MXL schema validates the user-defined model semantics. The RUBE_QM framework is introduced in the next chapter.

4.3 RUBE_QM Frameworks

4.3.1 Event Scheduling

Executing a system model sequentially by the time slicing method and event scheduling method is used in the simulation field based on the von Neumann-style architecture definition [24]. The time slicing method and the event scheduling method use a global clock to control events. The main difference between the two methods is how the events are executed using time. The time slicing method increases the given amount of time to the global clock value. It executes the event that matches the event time with the global clock time. In the event scheduling method, events are executed when the event time is the minimum time on the future event list. Events in the event list are sorted by the non-decreasing order of the system entering time and stored. The global clock time is then set by that event time so the globe time is increased irregularly. The time slicing method has the problem of encoding a delay between the model elements [24]. RUBE_QM uses the event scheduling method for simulation because the interarrival times are irregular

4.3.2 RUBE_QM Frameworks

The RUBE_QM architecture has the same translation steps as RUBE. However, certain steps perform additional works. The dynamic modeling and simulation in RUBE_QM is achieved by decoupling the model into two components: the model representation (a scene file) and the model semantics (model files). The model representation and the model semantics are defined using geometry modeling language (VRML and SVG) and MXL model topology, respectively. We will discuss the model representation file and MXL in detail in the next chapter.

The model file in RUBE_QM is an abstract description of a heterogeneous model semantic. Figure 4-3 shows an example in which the user defines a model using a source element, a facility element containing a queue and two servers, and a sink element. As shown in Figure 4-3, the elements have different shapes and model definitions.

RUBE_QM translates the model file into a homogenous block diagram file and creates JavaScript using the model translator, MXL2DXL.xsl, written in XML XSLT. Figure 4-3 shows the 2D presentation of the queuing model, and the corresponding DXL model is shown in Figure 4-4.

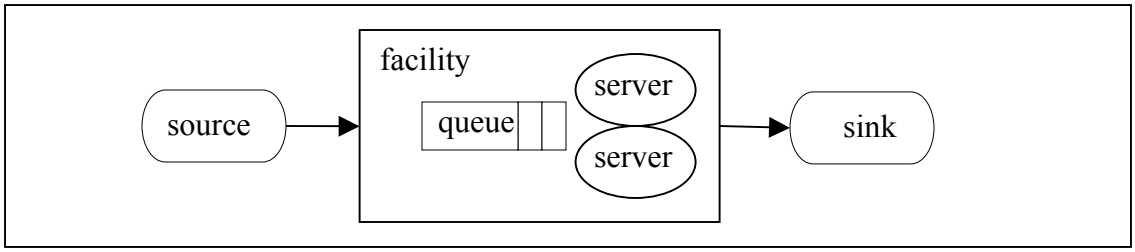

Figure 4-3. A 2D example of a queuing model file in MXL

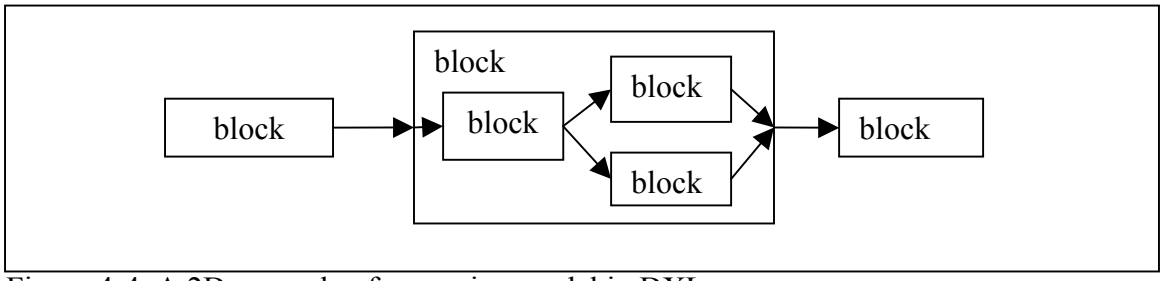

Figure 4-4. A 2D example of a queuing model in DXL

DXL is a low level (assembly level) modeling language [12]. It represents a multimodel containing more than one heterogeneous sub-model type within a homogeneous block diagram [16]. When the queuing model files are translated into a DXL level block diagram, RUBE_QM inserts block control behavior elements into the DXL block diagram and creates JavaScript files. For example, RUBE_QM creates an event generation JavaScript file, while the **source** element is translated into a DXL level

modeling language. To create an event generation JavaScript file, MXL2DXL.xsl uses a random number generation function provided in SimPackJ/S [24]. The SimPackJ/S is a library of simulation programs written in Java and JavaScript. When the **facility** element is translated into a DXL level block diagram, RUBE_QM creates a **queue** and **server** element control JavaScript files user-defined at MXL model file. We will discuss how to translate MXL to DXL in detail in the next chapter.

After the DXL level homogeneous block model is generated, RUBE_QM translates this model file into an executable JavaScript file using Java DOM, DXL.class. The DXL.class merges the user-created JavaScript file and the RUBE_QM-created JavaScript files into a single JavaScript file. MFE integrates the scene file in X3D or SVG with the RUBE_QM-generated JavaScript code and produces a dynamic model file, which contains the model dynamics in X3D or SVG. This RUBE_QM MFE has an identical functionality with the MFE in RUBE. In order to convert into a Web-based 3D simulation and model file, RUBE_QM uses file converters to transform X3D into VRML. This file is provided as an open source. The Model Fusion Engine (MFE) in RUBE_QM will be discussed in detail in Chapter 6.

CHAPTER 5
MODELING IN RUBE_QM

RUBE_QM is a framework for modeling and simulation in a queuing model. RUBE divides modeling into a representation file and a model semantic file. The RUBE_QM maintains this main principle. In this chapter, modeling in RUBE_QM is discussed in detail.

5.1 MXL Model Structure

The model semantic file (model file) of the RUBE_QM defines the model structure and dynamics of the model. The model file is defined in Multimodel eXchange Language (MXL) which is written in XML. Up to this point, RUBE MXL provides the effective way in presenting model topology, the dynamic behaviors, and the simulating information of the FBM (Function Block Model) and FSM (Finite State Machine). The RUBE_QM architecture provides model topology and the semantic of the queuing model using XML while maintaining the support of FBM and FSM. RUBE_QM is XML-based and it can easily extend its boundaries while maintaining its current support.

The main difference between FBM and FSM and the queuing model is that the queuing model events may be processed asynchronously. Events in FBM and FSM are processed without delay but the queuing model events may be stored and delayed for a given amount of time. The model translation steps in RUBE_QM are made to provide asynchronous and synchronous event simulation. Basically, the main structure of RUBE MXL supports asynchronous event simulation.

Model files are defined using XML, so XML terms are used here to explain model topology. The schema of MXL, which defines the model topology and semantics in RUBE_QM is provided in Appendix A. The topology of the model definition elements is shown in 2D diagrams in Figure 5-1.

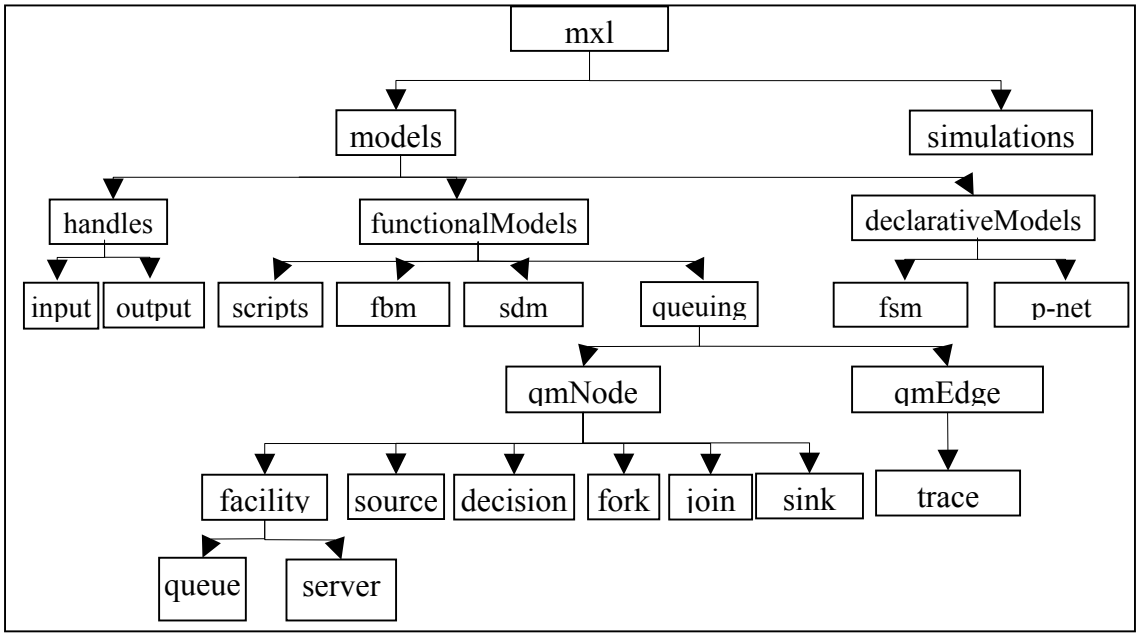

Figure 5-1. Model topology in MXL

The type attribute of model elements defines the type of the model. The model elements are represented by model node elements. The model edge elements describe the connectivity between the node elements. Some node elements have script elements as sub-elements. Script elements have an **id** attribute represented in the JavaScript file URI containing the dynamic behaviors of the node or edge elements. The **func** attributes in the Script element indicate the name of the function and contain the model dynamics in the JavaScript file. The **id** attribute of the node elements refers to the unique name of the element among all the MXL model files. The simulation element contains information needed for simulating the model that is already defined. These model topology elements

can be defined recursively using the XML technology, which makes it possible to define multimodel modeling and simulation.

For example, the queuing system, which consists of one CPU and four Disks, is used from the Modeling and Simulation class (CAP 4800 or CAP 5805) textbook. Each facility has one queue and one server.

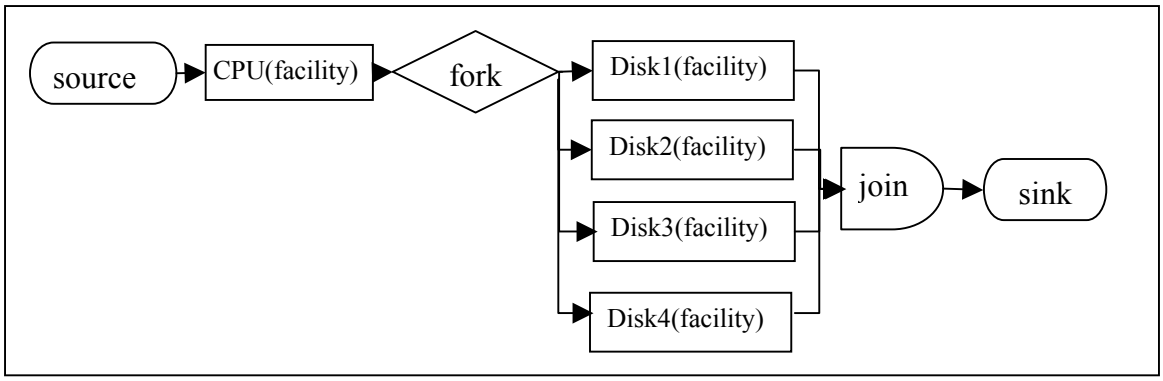

Figure 5-2. 2D diagram of CPU Disk queuing model

The 2D queuing model diagram, which is composed of a **source**, a **fork**, a **sink,** and five **facility** nodes, is shown in Figure 5-2. The identical representation of the MXL definition for the queuing model file is given in Figure 5-3.

```
<?xml version="1.0" encoding="UTF-8"?>
<MXL>
    <queuing id="CPU_DISK">
        <source id="F1">
            <entity id="job" generationFunction="uniform" parameter_1="0" parameter_2="10"/>
            <output id="jobF1" datatype="String" index="0"/>
            <script lang="JavaScript" src="Qnet.js" func="Generator"/>
        </source>
        <facility id="CPU">
            <input id="jobFromF1" datatype="String" index="0"/>
            <output id="jobCPU1" datatype="String" index="0"/>
            <queue id="queue" serviceTime="3">
          <script lang="JavaScript" src="Qnet.js" func="cpu"/>
```

Figure 5-3. MXL definition of one CPU four Disks queuing model

```
            </queue>
            <server id="server">
                <script lang="JavaScript" src="Qnet.js" func="cpuserver"/>
            </server>
        </facility>
        <fork id="F2">
            <input id="jobFromCPU1" datatype="String" index="0"/>
            <output id="jobF2_1" datatype="String" index="0" rate="0.25"/>
            <output id="jobF2_2" datatype="String" index="0" rate="0.25"/>
            <output id="jobF2_3" datatype="String" index="0" rate="0.25"/>
            <output id="jobF2_4" datatype="String" index="0" rate="0.25"/>
            <distribution id="F2_dis" distributionFunction="uniform" parameter_lowerBound="0"
parameter_upperBound="10"/>
        </fork>
        <facility id="DISK1">
            <input id="jobFromF2_1" datatype="String" index="0"/>
            <output id="jobF3_1" datatype="String" index="0"/>
            <queue id="queue1" serviceTime="3">
                <script lang="JavaScript" src="Qnet.js" func="disk1"/>
            </queue>
            <server id="server1">
                <script lang="JavaScript" src="Qnet.js" func="disk1server"/>
            </server>
        </facility>
        <facility id="DISK2">
            <input id="jobFromF2_2" datatype="String" index="0"/>
            <output id="jobF3_2" datatype="String" index="0"/>
            <queue id="queue2" serviceTime="3">
                <script lang="JavaScript" src="Qnet.js" func="disk2"/>
            </queue>
            <server id="server2">
                <script lang="JavaScript" src="Qnet.js" func="disk2server"/>
            </server>
        </facility>
        <facility id="DISK3">
            <input id="jobFromF2_3" datatype="String" index="0"/>
          <output id="jobF3_3" datatype="String" index="0"/>
```

Figure 5-3 Continued

```
            <queue id="queue3" serviceTime="3">
                <script lang="JavaScript" src="Qnet.js" func="disk3"/>
            </queue>
            <server id="server3">
                <script lang="JavaScript" src="Qnet.js" func="disk3server"/>
            </server>
        </facility>
        <facility id="DISK4">
            <input id="jobFromF2_4" datatype="String" index="0"/>
            <output id="jobF3_4" datatype="String" index="0"/>
            <queue id="queue4" serviceTime="3">
                <script lang="JavaScript" src="Qnet.js" func="disk4"/>
            </queue>
            <server id="server4">
                <script lang="JavaScript" src="Qnet.js" func="disk4server"/>
            </server>
        </facility>
        <join id="F4" collectionSize="4">
            <input id="jobFromF3" datatype="String" index="0"/>
            <output id="jobF4" datatype="String" index="0"/>
        </join>
        <sink id="F5">
            <input id="jobFromF4" datatype="String" index="0"/>
            <script lang="JavaScript" src="Qnet.js" func="sink"/>
        </sink>
        <trace from="jobF1" to="jobFromF1"/>
        <trace from="jobCPU1" to="jobFromCPU1"/>
        <trace from="jobF2_1" to="jobFromF2_1"/>
        <trace from="jobF2_2" to="jobFromF2_2"/>
        <trace from="jobF2_3" to="jobFromF2_3"/>
        <trace from="jobF2_4" to="jobFromF2_4"/>
        <trace from="jobF3_1" to="jobFromF3"/>
        <trace from="jobF3_2" to="jobFromF3"/>
        <trace from="jobF3_3" to="jobFromF3"/>
        <trace from="jobF3_4" to="jobFromF3"/>
        <trace from="jobF4" to="jobFromF4"/>
        </queuing>
    <simulation start_time="0.0" end_time="100" delta_time="1" cycle_time="1"/>
</MXL>
```

Figure 5-3. Continued

The **source** node includes one **entity,** one **output,** and one **script** element in the MXL definition file. The **entity** element in the **source** node is used in the model translation for creating the event generation function in JavaScript. The attributes of the entity node **generationFunction, parameter_1, parameter_2** are used to select the interarrival time generation function. Several random number generation functions are provided in SimPackJ/S, and RUBE_QM uses these functions. The **facility** element is composed of one **queue**, one **server**, an **input**, and an **output** node. One **facility** can have more than one **server** node, but only one **queue** node per **facility** node is possible. In the example MXL, the five facilities are defined as one CPU and four Disks. The **queue** node, a sub-node of **facility**, creates an FIFO queue and assigns an index to control the queuing model in JavaScript. The **serviceTime**, the attribute of the **queue** node, defines how long it takes to serve an incoming event at the server. This service time can also be defined using the random number generation function provided in SimPackJ/S. The **server** element has a **script** element in which the user can assign the behavior components to the server model presentation. The **fork** element is used to pick a single output using a cumulative distribution generated from the probability attached to each **output** element. The attribute **rate** is used in the **fork** element to pick the next output destination. The **join** element is used to adjust the event sequence. The attribute **collectionSize** defines the number of events to be arranged in the sequence at one time. The **sink** element is used to destroy the incoming event and update the processed event record.

The **trace** element connects the two elements: one from the elements that **id** of **output** sub-element is equal to **from** attribute of **trace,** and the other from the elements

that **id** of **input** sub-element is equal to **to** attribute of **trace**. The type of **data_type** of transporting element should be identical.

The **simulation** element defines information about the simulation execution. The **simulation** element has **start_ time, end_time, cycle_time,** and **delta_time. Start_time** and **end_time** signify when the simulation should start and finish. The **delta_time** defines the interval between events in the FBM and FSM model definition. However, the queuing model definition employs it to define the interarrival time of entities or guests. Closely investigated, the facility node has no **trace** element to connect between the **queue** and the counter **server** element. These traces between the queue and server are automatically generated at the model translation step. In the next chapter, we will look into the translation step. Figure 5-4 shows the queuing model elements and attributes.

Node	Attribute	Element
<Source>	<id: String> <Interarrival time: unsigned Integer> <output>	<input> <output> <script>
<facility>	<id: String>	<input> <output> <queue> <server>
<queue>	<id: String> <Size: Integer, default = unbound>	<script>
<server>	<id: String> <Service Time: unsigned Integer >	<script>
<Decision>	<id: String>	<script> <input> <output>
<fork>	<id: String> <distributionFunction> <distributionParameter>	<input> <output>

Figure 5-4. Queuing model elements and attributes

<sink>	<id:String>	<script>
<block>	<id:String>	<input> <output> <script>
<simulation>	<start_time:float> <end_time:float> <cycle_time:float> <delt_time:float>	
<trace>	<from:String> <to:String>	

Figure 5-4 Continued

5.2 JavaScript Function in RUBE_QM

RUBE supports the dynamic presentation of a model employing user-defined JavaScript functions. JavaScript functions, located in separate files, are referenced by **id** and **func** attributes of a **script** element of the model definition. Model behavior functions for a CPUDisk example are shown in Figure 5-5.

```
Function Generator(){
   return id;
}
function cpu(block_token){
   return (block_token.attr[3]);
}
function cpuserver(block_token){
   return (block_token.attr[3]);
}
function disk1(block_token){
   return (block_token.attr[3]);
}
function disk1server(block_token){
   return (block_token.attr[3]);
}
function disk2(block_token){
   return (block_token.attr[3]);
}
function disk2server(block_token){
   return (block_token.attr[3]);
}
```

```
function disk3(block_token){
   return (block_token.attr[3]);
}
function disk3server(block_token){
   return (block_token.attr[3]);
}
function disk4(block_token){
   return (block_token.attr[3]);
}
function disk4server(block_token){
   return (block_token.attr[3]);
}
function decision(){
   this.output[0]='decision()';
}
function sink(block_token){
   return (block_token.attr[3]);
}
```

Figure 5-5. Qnet.js: user-defined model behavior file

The user-defined model behavior functions are copied and pasted into the system-generated JavaScript file in the translation sequence. Pasted behavior files are invoked at the appropriate position to the RUBE_QM-generated JavaScript file. When queuing elements are translated to the DXL level model file, RUBE_QM generates appropriate JavaScript functions and files to control the queuing model event simulation.

When **source** elements are translated into the corresponding homogeneous block model, RUBE_QM generates a JavaScript file to create events or entities as in the user-defined interarrival time in the MXL model file. User-defined JavaScript functions are invoked in the RUBE_QM-generated JavaScript function to support the model dynamics. RUBE_QM uses the object, **block_token**, to support the asynchronous event processing inside of a system. The **block_tokens** carry the event id, entering time, and the output of the current model behavior function. Figure 5-6 shows the segment code of the JavaScript code, which is converted, from the user-defined model elements and JavaScript.

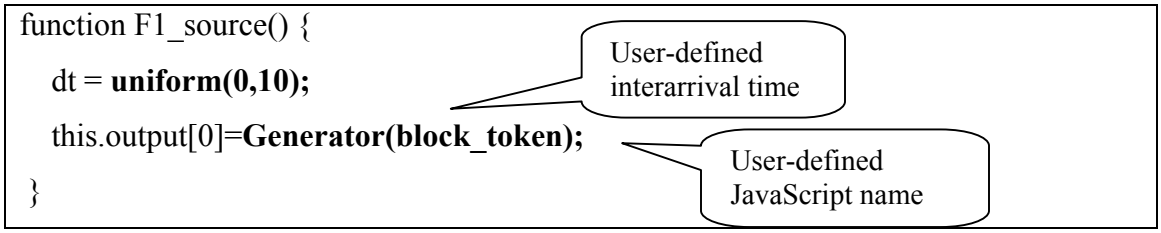

Figure 5-6. source_F1.js

The **dt** in Figure 5-5 is the interarrival time used to generate the next event. The **facility** is translated into the homogeneous block model while creating several model components. First, the model translation process creates an FIFO queue data structure and a control function provided in SimPackJ/S. Second, the connection elements between the queue block and the server blocks are created without user definition. Figure 5-6 shows the output file written in JavaScript using the queuing model definition in MXL.

```
function Q_request() {
  var server_num=1;
if(block_inform[2])
{
   queue_id[0] = create_facility("CPU",1);
   block_inform[2]=false;
}
if(!request(queue_id[0],block_token,0)){
   delay[3]= 3;
   this.output[0]=cpu(block_token);
   }
}
```

Queue data structure id

Invoke user-defined JavaScript

Figure 5-7. request_Q.js

When the **server** element is changed into the corresponding model block, RUBE_QM creates a **dequeue** function, a release function, and invokes the user-defined function. Figure 5-8 shows a JavaScript file when the **server** element transforms into a homogeneous block model.

```
Function Q_release() {
release(queue_id[0],block_token);
this.output[0]=cpuserver(block_token);
}
```

invoke dequeue JavaScript

Invoke user - defined JavaScript

Figure 5-8. release_Q.js

When a **fork** element is translated into a homogeneous block model, RUBE_QM creates the following JavaScript file during transformation. RUBE_QM uses the random number generation function, which the user may specify as the function name in the MXL model definition, as shown in Figure 5-9

```
Function F2_fork() {
 var count, tmp, tmp2;
 var lowerBound= 0;
 var upperBound= 10;
 branch_prob[0]=0.25;
```

Figure 5-9. fork_F2.js

```
branch_prob[1]=branch_prob[2-2] + 0.25;
branch_prob[2]=branch_prob[3-2] + 0.25;
branch_prob[3]=branch_prob[4-2] + 0.25;
tmp = uniform(lowerBound,upperBound);
tmp = tmp/(upperBound + lowerBound)
count = 0;
while (branch_prob[count]<tmp) count++;
   this.output[count] = this.input[0];
}
```

Invoke user-defined random number generation function JavaScript

Figure 5-9 Continued

The **join** model definition needs to create a data structure to save the events and the sorting function. For this, RUBE_QM creates an FIFO queue data structure and invokes the sorting function, which is provided in SimPackJ/S as well. The **sink** block model needs to store the completed event data, so we need to invoke the **update_completions()** function. Figures 5-10 and 11 show the JavaScript generated from the transformation process.

```
Function F5_join() {
 if(first){
 queue_id[1] = create_facility("F5",1);
first=false;
 }
 an_item.time = current_time;
 for (i=0;i<MAX_NUM_ATTR;i++)
     an_item.token.attr[i] = block_token.attr[i];
 an_item.event = 7;
 if(collectionCount <3){
          insert_list(facility[queue_id[1]].queue,an_item,BEHIND_PRIORITY_KEY);
collectionCount++;
 }
 else{
while(collectionCount!=0){
```

Figure 5-10. join_F5.js

```
remove_front_list(facility[queue_id[1]].queue,an_item);
an_item.time = current_time;
heap_insert(an_item);
collectionCount--;
}
this.output[0]=this.input[0]; }
}
```

Figure 5-10. Continued

```
function F5_sink() {
   sink(block_token);
   update_completions();
 }
```

Invoke User-defined JavaScript

Figure 5-11. sink_F5.js

5.3 Scene File in RUBE_QM

RUBE supports separation between the model representation and model semantics. The functionality of RUBE provides the user freedom of selecting scene model objects. The user can User defines model geometry with little concern about model semantics.

For example, Avatars represent entities or guests, and each model node element is represented as follows: each **queue** node can be visualized as hexahedrons, each source and each server are represented in cylinders, and **fork** and **join** node elements are mapped into a box. Figure 5-12 shows the CPUDisk scene file in VRML with an Avatar. The VRML scene files can be created using 3-D geometry-authoring tools, such as *Cosmo world* or blender.

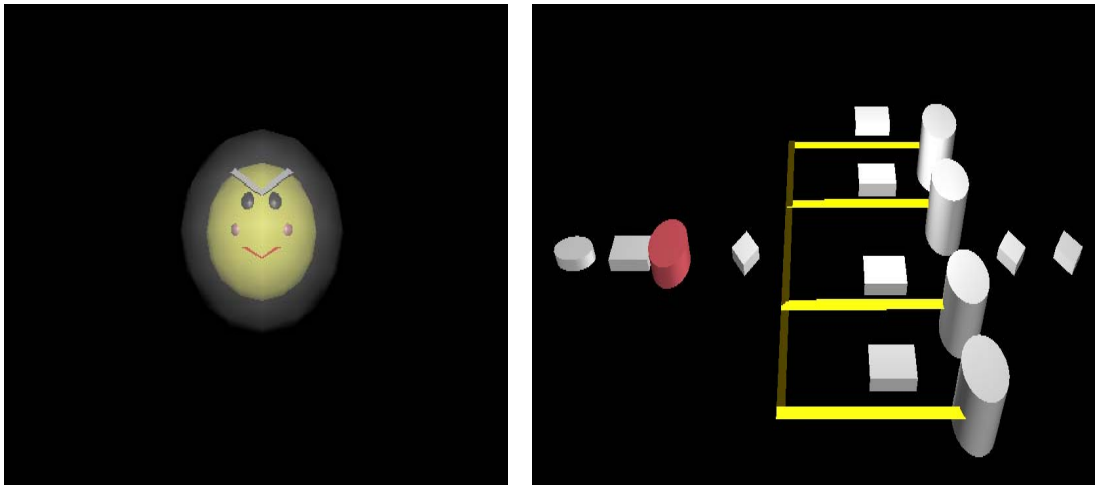

Figure 5-12. An Avatar and CPUDisk presentation

CHAPTER 6
TRANSLATION IN RUBE_QM

The RUBE_QM creates a scene file in which the user can add a visual dynamic behavior of the scene object in VRML with several translation processes. Each translation step is described in this chapter.

6.1 Overview

The model simulation file in RUBE is produced through several translation processes, mainly the translating XML files to the JavaScript files, creating the JavaScript files, and integrating scene files with JavaScript files. The RUBE_QM translation process is almost identical with RUBE, but contains a few differences in treating the MXL model files. Some MXL model elements need to be added and manipulated to build a queuing model simulation, as described in Chapter 5. The MXL model file, which is built in the MXL queuing model elements, needs to be validated by the MXL schema to check for correctness of the model file. After its correctness is checked, RUBE_QM starts the translation process. RUBE_QM translates heterogeneously defined model files into homogeneous block model files (DXL level block model) using MXL2DXL.xsl, which was discussed in Chapter 5. RUBE_QM translates the DXL level block model into JavaScript using DXL.class written in Java DOM, and at the same time the semantic correctness of MXL model file is checked during the translation process as well[12]. The JavaScript model file, merged with the scene file, is provided with connections between the model files and the scene objects. It uses MFE (Model Fusion Engine) written in XML XSLT, namely, MFE.xsl. Users can make dynamic scenes using the JavaScript and

VRML node after translating X3D into VRML using the X3D to VRML converter program in XML XSLT.

6.2 MXL to DXL Translation

The translation from a heterogeneous model file, MXL, to a homogeneous model file, DXL, is done by using the transformation rules defined in the XSLT stylesheet, MXL2DXL.xsl. The DXL is the intermediate layer between MXL and the executable JavaScript code. RUBE_QM translates every node element, defined in MXL, into a block element in DXL and every edge element into a connection element. Some edge elements are specifically added to provide the connection between the queue element and server element(s) without user definition. RUBE_QM follows the transformation steps of RUBE to maintain compatibility with the original RUBE and to support multimodeling capabilities. Figure 6-1 shows the DXL level CPUDisk queuing model file, as previously described. The RUBE_QM DXL level file has different elements from the original RUBE DXL level file. The **definition_in** element is added to store the user-defined model behavior JavaScript file, and it will be invoked inside of the translator-created JavaScript function. The **func** value will be inserted into JavaScript files and also invoked as explained in the previous chapter. For example, the **Generator** function is invoked in the **F1_source** JavaScript function, as shown in Figures 6-1 and 5-6.

The **facility** element will be translated as a heterogeneous model definition, which contains several model definitions. When the **facility** element is translated into a block model, RUBE_QM makes a link between the **queue** and **server** elements with other elements in the facility through **input** and the **output** elements. Figure 6-2 shows the conversion of the facility model definition to a DXL level block diagram.

```
<DXL>
    <block id="F1" type="SYNC">
        <port id="F1.jobF1" type="OUTPUT" target="CPU.jobFromF1" data_type="String"/>
        <definition_in id="Qnet.js" func="Generator" lang="JavaScript"/>
        <definition id="source_F1.js" func="F1_source" lang="JavaScript"/>
    </block>                                                    ,
                                    ,
                                    ,
    <block id="F5" type="SYNC">
        <port id="F5.jobFromF4" type="INPUT" source="F4.jobF4" data_type="String"/>
        <definition_in id="Qnet.js" func="sink" lang="JavaScript"/>
        <definition id="sink_F5.js" func="F5_sink" lang="JavaScript"/>
    </block>
    <connect id="jobF1TojobFromF1">
        <port id="jobF1TojobFromF1.OP1" type="INPUT" source="F1.jobF1"/>
        <port id="jobF1TojobFromF1.IP1" type="OUTPUT" source="CPU.jobFromF1"/>
    </connect>
    <connect id="jobCPU1TojobFromCPU1">
        <port id="jobCPU1TojobFromCPU1.OP1" type="INPUT" source="CPU.jobCPU1"/>
        <port id="jobCPU1TojobFromCPU1.IP1" type="OUTPUT" source="F2.jobFromCPU1"/>
    </connect>
    <simulation start_time="0.0" end_time="100" delta_time="1" cycle_time="1"/>
</DXL>
```

Figure 6-1. A DXL segment of CPUDisk model

```
<block id="CPU" type="ASYNC">
        <port id="CPU.jobFromF1" type="INPUT" source="F1.jobF1" target="queue.jobFromCPU"
         data_type="String"/>
        <port id="CPU.jobCPU1" type="OUTPUT" target="F2.jobFromCPU1" data_type="String"/>
        <DXL id="CPU">
            <block id="queue" type="ASYNC">
                <port id="queue.jobFromCPU" type="INPUT" source="CPU.jobFromF1"
                    data_type="String"/>
                <port id="queue.jobqueueserver" type="OUTPUT" target="server.jobFromqueueserver"
                    data_type="String"/>
                <definition_in id="Qnet.js" func="cpu" lang="JavaScript"/>
                <definition id="request_queue.js" func="queue_request" lang="JavaScript"/>
            </block>
            <block id="server" type="ASYNC">
                <port id="server.jobFromqueueserver" type="INPUT" source="queue.jobqueueserver"
                 data_type="String"/>
                <port id="server.jobserver" type="OUTPUT" target="CPU.jobCPU1"
                 data_type="String"/>
                <definition_in id="Qnet.js" func="cpuserver" lang="JavaScript"/>
                <definition id="release_queue.js" func="queue_release" lang="JavaScript"/>
            </block>
            <connect id="jobFromF1TojobFromCPU">
             <port id="jobFromF1TojobFromCPU.OP1" type="INPUT" source="CPU.jobFromF1"/>
            <port id="jobFromF1TojobFromCPU.IP1" type="OUTPUT" source="queue.jobFromCPU"/>
            </connect>
            <connect id="jobqueueserverTojobFromqueueserver">
                <port id="jobqueueserverTojobFromqueueserver.OP1" type="INPUT"
                 source="queueserver.jobqueueserver"/>
                <port id="jobqueueserverTojobFromqueueserver.IP1" type="OUTPUT"
                 source="server.jobFromqueueserver"/>
            </connect>
            <connect id="jobserverTojobCPU1">
                <port id="jobserverTojobCPU1.OP1" type="INPUT" source="server.jobserver"/>
                <port id="jobserverTojobCPU1.IP1" type="OUTPUT" source="CPU.jobCPU1"/>
            </connect>
        </DXL>
</block>
```

Figure 6-2. A DXL segment of CPUDisk facility

The **trace** elements in MXL are converted into a connection element in the DXL block model. The connection elements in DXL can be described as a bridge which links the input and output ports. The DXL can be viewed as a directed graph of blocks with ports and connections between the blocks. Each block in DXL is associated with one or more input and output ports. Figure 6-3 shows a CPUDisk queuing model in the DXL diagram.

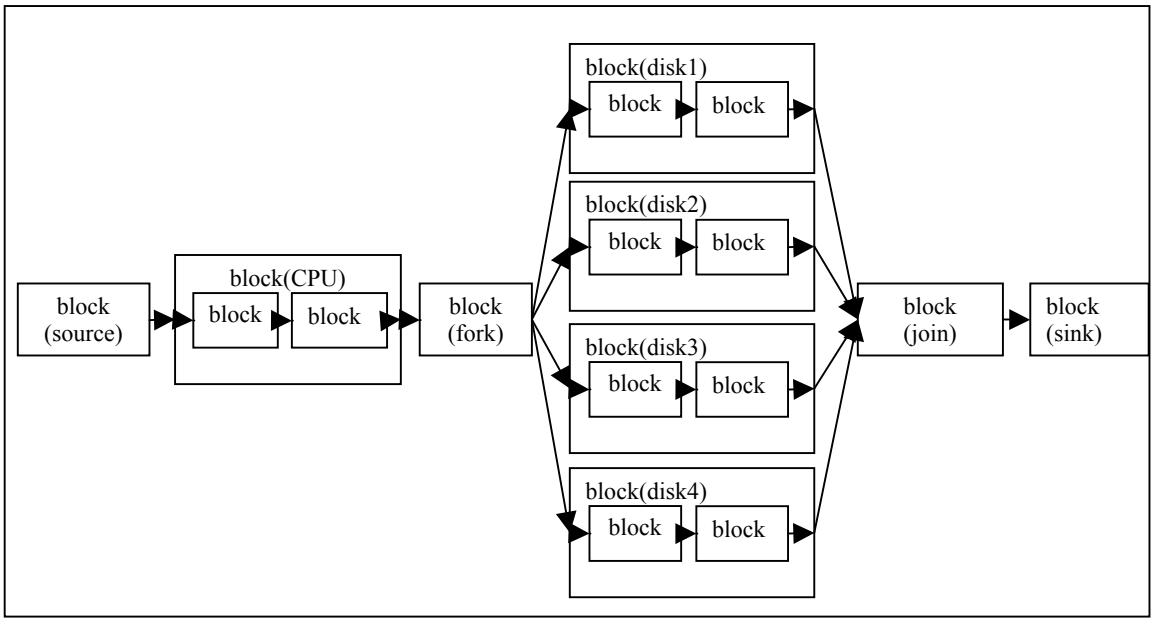

Figure 6-3. Queuing model DXL diagram for CPUDisk

By simplifying the representation of the model topology, DXL can easily be translated into the JavaScript code and support the multimodel modeling and simulation.

6.3 DXL to JavaScript Translation

The final executable JavaScript is produced by DXL to a JavaScript translation procedure. User-defined JavaScript and MXL2DXL.xsl translator-created JavaScript are fused and manipulated by DXL to the JavaScript translator (DXL.class), which is written in Java DOM. The DXL.class also uses SimPackJ/S to produce the final executable JavaScript model file. SimPackJ/S provides a simulation frame program, such as event

handling and event creating at RUBE. RUBE_QM uses SimPackJ/S codes to provide an asynchronous event handling function for the queuing model simulation. The output of DXL.class JavaScript file can be divided into four parts: 1) copied program from SimPackJ/S; 2) copied program from user-defined JavaScript; 3) code from MXL to DXL translator-generated based on the model topology; and 4) code from translator-generated JavaScript.

In the RUBE_QM queuing model simulation, JavaScript files and data structure are added for event controlling. The event can occur at various interval times from when the last event happened, and events need to be served at various times. Due to this fact, the event control mechanism in RUBE_QM needs the capability of manipulating asynchronous events while maintaining synchronous event handling mechanisms. When we run the JavaScript simulation code, RUBE_QM produces results from each model node element in the MXL model definition if the node element contains output elements as its sub-element. Every MXL node element is converted into the JavaScript class and its sub-elements contain the variables in its class. These variable values are sent to the scene file to add dynamics to the corresponding scene object using the VRML structure, such as ROUTE, Script, and PROTO. Figure 6-4 shows segments of the JavaScript file and VRML file, which sends and receives variable values using the Script function.

The JavaScript Simulation function in RUBE_QM, update_block(), controls the event handling mechanism, and the dynamic variable values are sent to the VRML structure at this translation step. The variable name **out_<node_id>** is the sender of the behavior value and the **rube.out_<node_id>** is the corresponding receiver of the value. Model Fusion Engine (MFE) using MFE.xsl in XSLT accomplishes this connection.

Simulation JavaScript	VRML File

function update_block(){
switch (event.value) {
case 0: ....
out_F1 = F1_object.output[0];
....
case 2: .....
out_CPU_queue = queue_object.output[0];
....
case 3:....
out_CPU_server = server_object.output[0];
....
case 6: ....
out_DISK1_queue1 = queue1_object.output[0];
....
case 7: ....
out_DISK1_server1 = server1_object.output[0];
....
case 18:....
}
}
	ROUTE rube.out_F1 TO
	user_Driver.modify_out_F1
	…
	ROUTE rube.out_CPU_queue TO
	user_Driver.modify_out_CPU_queue
	…
	ROUTE rube.out_CPU_server TO
	user_Driver.modify_out_CPU_server
	…
	ROUTE rube.out_DISK1_queue1 TO
	user_Driver.modify_out_DISK1_queue1
	…
	ROUTE rube.out_DISK1_server1 TO
	user_Driver.modify_out_DISK1_server1
	…

Figure 6-4. Simulation JavaScript and VRML file

6.4 Model Fusion Engine and Translation from X3D to VRML

Model Fusion Engine (MFE) in RUBE_QM is identical to the 3D Model Fusion Engine of RUBE. The Model Fusion Engine combines the 3D scene file with the model behavior JavaScript, and it produces a model scene containing the dynamics in X3D. The VRML Script nodes allow creating node behaviors through a programming language such as Java or Java/ ECMAScript. RUBE_QM uses VRML Script nodes to control the scene dynamics [6]. The MFE finds every model node, which contains an output node as a sub-element from the MXL model definition and makes a connection with the scene file. The MFE produces the same scene with Head Up Display (HUD), which supports simulation control functions and contains the dynamic inside of the scene file in X3D. RUBE_QM uses converters to transform X3D to VRML in XSLT, namely, NistX3d2Vrml.xsl. Figure 6-5 shows the final output VRML file segments of

RUBE_QM. The VRML file can be divided into the following segments: 1) copied code from the model scene; 2) script nodes which link user-defined model dynamics with the VRML model object; and 3) display control code segment.

Segment of VRML scene file	Explain
DEF CPU Transform {} DEF DISK1 Transform {}	Copy of scene code from user definition
DEF rube Script { eventIns for the HUD eventOut SFTime value_changed … eventOut SFString out_F1 … eventOut SFString out_CPU_queue eventOut SFString out_CPU_server … url "rubeDXL.js"}	RUBE_QM make bridge with user-defined model behavior script output and RUBE_QM generate Script output using VRML Script node.
DEF rube_Clock TimeSensor {…} EXTERNPROTO rube_Display […] "rube_Display.wrl#rube_Display" EXTERNPROTO rube_Slider [ ] "rube_Display.wrl#rube_Slider" DEF rube_HUD rube_Display { …}	RUBE generate display controller using VRML PROTO node (Head Up Display)
DEF user_Driver Script { … url "javascript: function modify_out_F1(s, ts){} … function modify_out_CPU_queue(s, ts){ } function modify_out_CPU_server(s, s){ } …"}	RUBE provides Script for user can add dynamic to the scene model object with the output of user-defined model behavior JavaScript.
ROUTE rube.out_F1 TO user_Driver.modify_out_F1 … ROUTE rube.out_CPU_queue TO user_Driver.modify_out_CPU_queue … ROUTE rube.out_CPU_server TO user_Driver.modify_out_CPU_server	Receive model output from user-defined behavior JavaScript and link to RUBE scene control variable (VRML variable).

Figure 6-5 Final RUBE_QMOutput segment

6.5 Model Simulation

The VRML file with dynamics has been created by RUBE_QM, as shown in Figure 6-5. RUBE links outputs of user-defined dynamics with the VRML Script nodes, such as **user_Driver**, using the VRML ROUTE structure. Many dynamic value control functions are in **user_Driver**. Many functions are defined using the **event_In** variable names inside of a **user_Driver** Script node. Each Script function manipulates the input of the function to control the dynamics of the scene and is sent out using the **event_Out** variable name. Figure 6-6 shows the example of a VRML Script node function. This function executes JavaScript print commands and output values using a **modified_out_CPU_queue[]** vector node.

```
function modify_out_CPU_queue(s, ts) {
  if(modify_out_CPU_queue!=null){
  print('CPU_queue(vrml):'+s);
  modified_out_CPU_queue[0]=s;
  modified_out_CPU_queue[1]=2;
  modified_out_F1[2]='CPU_queue';
   }
  }
```

Figure 6-6. Example of a VRML Script node

Figure 6-7 is a snapshot of a VRML file being executed. Avatars represent guests or entities and each object represents a **queue, CPU, Disk, fork, Join**, and **Sink**, respectively. As simulation time increases, the Avatars get into the CPUDisk System and move their position at a given time.

Figure 6-7 shows that guest 0 already has finished his job at the system. Guest 1, 2 and 3 are waiting for Guest 4 at the join model representation while Guest 4 is being

serviced at the Disk 4 server. The CPU server is serving Guest 5, and Guests 6, 7, and 8 are waiting for their turn at the waiting line (CPU queue).

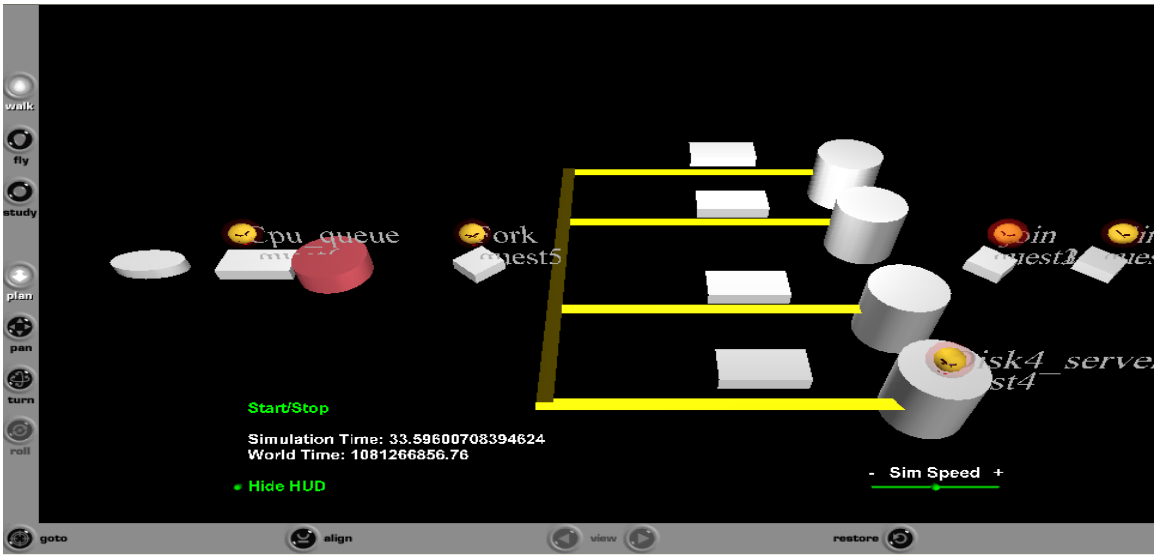

Figure 6-7. A snapshot of CPUDisk VRML file with dynamic

CHAPTER 7
CONCLUSIONS

The goal of this research is to expand the supporting model type of RUBE to the queuing models while maintaining the current support for the FBM and FSM model types.

To add new frameworks to the RUBE architecture, newly inserted frameworks have to be compatible with the methodology of RUBE and support the main goal of RUBE: construct an easily personalizable and customizable multimodel with Web-based visualization.

RUBE_QM supports the modeling and simulation of the queuing system. To be compatible with RUBE, RUBE_QM adopts the methodology of RUBE.

To develop frameworks for the new model type, RUBE_QM needs to add new model elements and semantics into MXL (Multimodel eXchange Language) in XML. The XML document handling mechanism is used along with the name correspondence. Therefore MXL easily expands its model type definition.

RUBE_QM creates data structures, which store events for some amount of time inside of the simulation environment, to support an asynchronous event in the queuing system. The event scheduling method is used to execute sequential events, which are in the future event list with the minimum time tag. The future event list stores events, which have to be executed, by the non-decreasing order of time. The event scheduling method is also used in RUBE.

The RUBE_QM translator generates files to support the following queuing model event handling mechanisms: 1) generates events at random interarrival time; 2) creates connection elements between the queue and the server(s); 3) distributes events as probability; 4) stores events until a given amount of entities arrived and sends an entity or entities to the next model element; and 5) finally destroys the elements.

I accomplished my research goal, which expands the supporting model of RUBE to queuing model while maintaining the methodology of RUBE.

The XML-based technologies are used as leverage in expanding the application boundary of RUBE. Since RUBE was built base on the XML-based technologies, the application boundary of RUBE can be extended to other model types as the research progress.

APPENDIX A
SCHEMA FOR MXL

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xsd:annotation>
    <xsd:documentation>Multi-Model Exchange Language (MXL) schema definition.  Copyright 2003 University of Florida.  All rights reserved.</xsd:documentation>
  </xsd:annotation>
  <xsd:simpleType name="fieldTypeName">
    <xsd:annotation>
      <xsd:appinfo>All supported data types should be placed in this list.  Future data types should be appended to the end of the list.</xsd:appinfo>
      <xsd:documentation>Enumerates all MXL data types.</xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="Boolean">
        <xsd:annotation>
          <xsd:appinfo>Boolean is analogous to xsd:boolean type.</xsd:appinfo>
          <xsd:documentation>Boolean is used to represent a logical type with possible values (true | false) to match the XML boolean type.</xsd:documentation>
        </xsd:annotation>
      </xsd:enumeration>
      <xsd:enumeration value="Booleans">
        <xsd:annotation>
          <xsd:appinfo>Booleans is analogous to xsd:boolean list type.</xsd:appinfo>
          <xsd:documentation>Booleans is used to represent an array of Boolean values.</xsd:documentation>
        </xsd:annotation>
      </xsd:enumeration>
      <xsd:enumeration value="Integer">
        <xsd:annotation>
          <xsd:appinfo>Integer is used analogous to xsd:integer type.</xsd:appinfo>
          <xsd:documentation>Integer is used to represent a 32-bit integer type.</xsd:documentation>
```

Figure A-1. MXL schema

```
                </xsd:annotation>
            </xsd:enumeration>
            <xsd:enumeration value="Integers">
                <xsd:annotation>
                    <xsd:appinfo>Integers is used analogous to xsd:integer list type.</xsd:appinfo>
                    <xsd:documentation>Integers is used to represent an array of Integer
values.</xsd:documentation>
                </xsd:annotation>
            </xsd:enumeration>
            <xsd:enumeration value="Float">
                <xsd:annotation>
                    <xsd:appinfo>Float is used analogous to xsd:float type.</xsd:appinfo>
                    <xsd:documentation>Float is used to represent a single-precision floating-point
type.</xsd:documentation>
                </xsd:annotation>
            </xsd:enumeration>
            <xsd:enumeration value="Floats">
                <xsd:annotation>
                    <xsd:appinfo>Floats is used analogous to xsd:float list type.</xsd:appinfo>
                    <xsd:documentation>Floats is used to represent an array of Float
values.</xsd:documentation>
                </xsd:annotation>
            </xsd:enumeration>
            <xsd:enumeration value="Floats">
                <xsd:annotation>
                    <xsd:appinfo>Floats is used analogous to xsd:float list type.</xsd:appinfo>
                    <xsd:documentation>Floats is used to represent an array of Float
values.</xsd:documentation>
                </xsd:annotation>
            </xsd:enumeration>
            <xsd:enumeration value="Double">
                <xsd:annotation>
                    <xsd:appinfo>Double is analogous to xsd:double type.</xsd:appinfo>
                    <xsd:documentation>Double is used to represent a double-precision floating-point
type.</xsd:documentation>
                </xsd:annotation>
            </xsd:enumeration>
            <xsd:enumeration value="Doubles">
```

Figure A-1. Continued

```
                <xsd:annotation>
                    <xsd:appinfo>Doubles is analogous to xsd:double list type.</xsd:appinfo>
                    <xsd:documentation>Doubles is used to represent an array of Double
values.</xsd:documentation>
                </xsd:annotation>
            </xsd:enumeration>
            <xsd:enumeration value="String">
                <xsd:annotation>
                    <xsd:appinfo>String is used analogous to an xsd:string type.</xsd:appinfo>
                    <xsd:documentation>String is used to represent a character string
type.</xsd:documentation>
                </xsd:annotation>
            </xsd:enumeration>
            <xsd:enumeration value="Strings">
                <xsd:annotation>
                    <xsd:appinfo>Strings is used analogous to an xsd:string list type.</xsd:appinfo>


                    <xsd:documentation>Strings is used to represent an array of String
values.</xsd:documentation>
                </xsd:annotation>
            </xsd:enumeration>
        </xsd:restriction>
    </xsd:simpleType>
    <xsd:element name="MXL">
        <xsd:annotation>
            <xsd:appinfo>The MXL file is broken into two main sections: Model Definition and
Simulation Frame.</xsd:appinfo>
            <xsd:documentation>This is the MXL root node.</xsd:documentation>
        </xsd:annotation>
        <xsd:complexType>
            <xsd:sequence>
                <xsd:group ref="models"/>
                <xsd:element name="simulation" minOccurs="0">
                    <xsd:complexType>
                        <xsd:attribute name="start_time" type="xsd:float" use="required"/>
                        <xsd:attribute name="end_time" type="xsd:float" use="required"/>
                        <xsd:attribute name="delta_time" type="xsd:float" use="required"/>
                        <xsd:attribute name="cycle_time" type="xsd:float" use="required"/>
```

Figure A-1. Continued

```
                    </xsd:complexType>
                </xsd:element>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:group name="models">
        <xsd:annotation>
            <xsd:appinfo>All model categories should be placed under the "choice" element.  New
model categories should be appended to the end of the current list.</xsd:appinfo>
            <xsd:documentation>Generic Model Group</xsd:documentation>
        </xsd:annotation>
        <xsd:sequence>
            <xsd:group ref="handles"/>
            <xsd:choice>
                <xsd:group ref="functionalModels"/>
                <xsd:group ref="declarativeModels"/>
            </xsd:choice>
        </xsd:sequence>
    </xsd:group>
    <xsd:group name="handles">
        <xsd:annotation>
            <xsd:documentation>Handle Group</xsd:documentation>
        </xsd:annotation>
        <xsd:sequence>
            <xsd:element name="input" minOccurs="0" maxOccurs="unbounded">
                <xsd:complexType>
                    <xsd:attribute name="id" type="xsd:ID" use="required"/>
                    <xsd:attribute name="datatype" type="fieldTypeName" use="required"/>
                    <xsd:attribute name="index" type="xsd:unsignedInt" use="required"/>
                </xsd:complexType>
            </xsd:element>
            <xsd:element name="output" minOccurs="0" maxOccurs="unbounded">
                <xsd:complexType>
                    <xsd:attribute name="id" type="xsd:ID" use="required"/>
                    <xsd:attribute name="datatype" type="fieldTypeName" use="required"/>
                    <xsd:attribute name="initial" type="xsd:NMTOKEN" use="optional"/>

                    <xsd:attribute name="index" type="xsd:unsignedInt" use="required"/>
```

Figure A-1. Continued

```
                    <xsd:attribute name="rate" type="xsd:double" use="optional"/>
                </xsd:complexType>
            </xsd:element>
        </xsd:sequence>
    </xsd:group>
    <xsd:group name="functionalModels">
        <xsd:annotation>
            <xsd:documentation>Functional Model Group</xsd:documentation>
        </xsd:annotation>
        <xsd:choice>
            <xsd:group ref="scripts"/>
            <xsd:group ref="fbm"/>
            <xsd:group ref="sdm"/>
            <xsd:group ref="queuing"/>
            <!--nklim-->
        </xsd:choice>
    </xsd:group>
    <xsd:group name="scripts">
        <xsd:annotation>
            <xsd:documentation>Script Group</xsd:documentation>
        </xsd:annotation>
        <xsd:choice>
            <xsd:element name="script">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element name="param" minOccurs="0" maxOccurs="unbounded">
                            <xsd:complexType>
                                <xsd:attribute name="value" type="xsd:NMTOKEN" use="required"/>
                                <xsd:attribute name="index" type="xsd:unsignedInt" use="required"/>
                                <!--<xsd:attribute name="name" type="xsd:token" use="required"/>-->
                            </xsd:complexType>
                        </xsd:element>
                    </xsd:sequence>
                    <xsd:attribute name="lang" use="required">
                        <xsd:simpleType>
                            <xsd:restriction base="xsd:string">
                                <xsd:enumeration value="JavaScript"/>
                                <xsd:enumeration value="Java"/>
                            </xsd:restriction>
```

Figure A-1. Continued

```
                    </xsd:simpleType>
                </xsd:attribute>
                <xsd:attribute name="src" type="xsd:anyURI" use="required"/>

                <xsd:attribute name="func" type="xsd:NCName" use="required"/>
            </xsd:complexType>
        </xsd:element>
    </xsd:choice>
</xsd:group>
<xsd:group name="fbm">
    <xsd:sequence>
        <xsd:element name="fbm">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:group ref="fbmNodes" minOccurs="0" maxOccurs="unbounded"/>
                    <xsd:group ref="fbmEdges" minOccurs="0" maxOccurs="unbounded"/>
                </xsd:sequence>
                <xsd:attribute name="id" type="xsd:ID" use="required"/>
                <xsd:attribute name="src" type="xsd:anyURI" use="optional"/>
            </xsd:complexType>
        </xsd:element>
    </xsd:sequence>
</xsd:group>
<xsd:group name="fbmNodes">
    <xsd:choice>
        <xsd:element name="block">
            <xsd:complexType>
                <xsd:choice>
                    <xsd:group ref="models"/>
                </xsd:choice>
                <xsd:attribute name="id" type="xsd:ID" use="required"/>
            </xsd:complexType>
        </xsd:element>
    </xsd:choice>
</xsd:group>
<xsd:group name="fbmEdges">
    <xsd:choice>
        <xsd:element name="trace" minOccurs="0">
            <xsd:complexType>
```

Figure A-1. Continued

```
                    <xsd:attribute name="from" type="xsd:IDREF" use="required"/>
                    <xsd:attribute name="to" type="xsd:IDREF" use="required"/>
                </xsd:complexType>
            </xsd:element>
        </xsd:choice>
    </xsd:group>
    <xsd:group name="sdm">
        <xsd:sequence>
            <xsd:element name="sdm">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:group ref="sdmNodes" minOccurs="0" maxOccurs="unbounded"/>
                        <xsd:group ref="sdmEdges" minOccurs="0" maxOccurs="unbounded"/>
                    </xsd:sequence>
                    <xsd:attribute name="id" type="xsd:ID" use="required"/>
                    <xsd:attribute name="src" type="xsd:anyURI" use="optional"/>
                </xsd:complexType>

            </xsd:element>
        </xsd:sequence>
    </xsd:group>
    <xsd:group name="sdmNodes">
        <xsd:choice>
            <xsd:element name="src">
                <xsd:complexType>
                    <xsd:attribute name="id" type="xsd:ID" use="required"/>
                </xsd:complexType>
            </xsd:element>
            <xsd:element name="sink">
                <xsd:complexType>
                    <xsd:attribute name="id" type="xsd:ID" use="required"/>
                </xsd:complexType>
            </xsd:element>
            <xsd:element name="rate">
                <xsd:complexType>
                    <xsd:choice>
                        <xsd:group ref="models"/>
                    </xsd:choice>
                    <xsd:attribute name="id" type="xsd:ID" use="required"/>
```

Figure A-1. Continued

```
                </xsd:complexType>
            </xsd:element>
            <xsd:element name="level">
                <xsd:complexType>
                    <xsd:choice>
                        <xsd:group ref="models"/>
                    </xsd:choice>
                    <xsd:attribute name="id" type="xsd:ID" use="required"/>
                    <xsd:attribute name="initial" type="xsd:double" use="required"/>
                </xsd:complexType>
            </xsd:element>
            <xsd:element name="aux">
                <xsd:complexType>
                    <xsd:choice>
                        <xsd:group ref="models"/>
                    </xsd:choice>
                    <xsd:attribute name="id" type="xsd:ID" use="required"/>
                </xsd:complexType>
            </xsd:element>
            <xsd:element name="const">
                <xsd:complexType>
                    <xsd:attribute name="id" type="xsd:ID" use="required"/>
                    <xsd:attribute name="value" type="xsd:double" use="required"/>
                </xsd:complexType>
            </xsd:element>
        </xsd:choice>
    </xsd:group>
    <xsd:group name="sdmEdges">
        <xsd:choice>
            <xsd:element name="arc">
                <xsd:complexType>

                    <xsd:attribute name="type" use="required">
                        <xsd:simpleType>
                            <xsd:restriction base="xsd:string">
                                <xsd:enumeration value="flow"/>
                                <xsd:enumeration value="cause-effect"/>
                            </xsd:restriction>
                        </xsd:simpleType>
```

Figure A-1. Continued

```
                    </xsd:attribute>
                    <xsd:attribute name="from" type="xsd:IDREF" use="required"/>
                    <xsd:attribute name="to" type="xsd:IDREF" use="required"/>
                </xsd:complexType>
            </xsd:element>
        </xsd:choice>
    </xsd:group>
    <xsd:group name="declarativeModels">
        <xsd:annotation>
            <xsd:documentation>Declarative Model Group</xsd:documentation>
        </xsd:annotation>
        <xsd:choice>
            <xsd:group ref="fsm"/>
            <xsd:group ref="p-net"/>
        </xsd:choice>
    </xsd:group>
    <xsd:group name="fsm">
        <xsd:all>
            <xsd:element name="fsm">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:group ref="fsmNodes" minOccurs="0" maxOccurs="unbounded"/>
                        <xsd:group ref="fsmEdges" minOccurs="0" maxOccurs="unbounded"/>
                    </xsd:sequence>
                    <xsd:attribute name="id" type="xsd:ID" use="required"/>
                    <xsd:attribute name="src" type="xsd:anyURI" use="optional"/>
                </xsd:complexType>
            </xsd:element>
        </xsd:all>
    </xsd:group>
    <xsd:group name="fsmNodes">
        <xsd:choice>
            <xsd:element name="state">
                <xsd:complexType>
                    <xsd:choice>
                        <xsd:group ref="models"/>
                    </xsd:choice>
                    <xsd:attribute name="id" type="xsd:ID" use="required"/>
                    <xsd:attribute name="start" type="xsd:boolean" use="optional" default="false"/>
```

Figure A-1. Continued

```
                </xsd:complexType>
            </xsd:element>
        </xsd:choice>
    </xsd:group>
    <xsd:group name="fsmEdges">

        <xsd:choice>
            <xsd:element name="transition">
                <xsd:complexType>
                    <xsd:choice>
                        <xsd:group ref="models"/>
                    </xsd:choice>
                    <xsd:attribute name="from" type="xsd:IDREF" use="required"/>
                    <xsd:attribute name="to" type="xsd:IDREF" use="required"/>
                </xsd:complexType>
            </xsd:element>
        </xsd:choice>
    </xsd:group>
    <xsd:group name="p-net">
        <xsd:all>
            <xsd:element name="p-net">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:group ref="p-netNodes" minOccurs="0" maxOccurs="unbounded"/>
                        <xsd:group ref="p-netEdges" minOccurs="0" maxOccurs="unbounded"/>
                    </xsd:sequence>
                    <xsd:attribute name="id" type="xsd:ID" use="required"/>
                    <xsd:attribute name="src" type="xsd:anyURI" use="optional"/>
                </xsd:complexType>
            </xsd:element>
        </xsd:all>
    </xsd:group>
    <xsd:group name="p-netNodes">
        <xsd:choice>
            <xsd:element name="condition">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element name="token" minOccurs="0" maxOccurs="unbounded">
```

Figure A-1. Continued

```
                        <xsd:complexType>
                            <xsd:attribute name="color" type="xsd:nonNegativeInteger"
use="optional" default="0"/>
                            <xsd:attribute name="init_tokens" type="xsd:nonNegativeInteger"
use="optional" default="0"/>
                        </xsd:complexType>
                    </xsd:element>
                    <xsd:choice>
                        <xsd:group ref="models" minOccurs="0"/>
                    </xsd:choice>
                </xsd:sequence>
                <xsd:attribute name="id" type="xsd:ID" use="required"/>
            </xsd:complexType>
        </xsd:element>
        <xsd:element name="event">
            <xsd:complexType>
                <xsd:choice>
                    <xsd:group ref="models"/>
                </xsd:choice>
                <xsd:attribute name="id" type="xsd:ID" use="required"/>
                <xsd:attribute name="time" type="xsd:double" use="optional" default="0.0"/>
            </xsd:complexType>
        </xsd:element>
    </xsd:choice>
</xsd:group>
<xsd:group name="p-netEdges">
    <xsd:choice>
        <xsd:element name="connection">
            <xsd:complexType>
                <xsd:choice>
                    <xsd:group ref="models" minOccurs="0"/>
                </xsd:choice>
                <xsd:attribute name="from" type="xsd:IDREF" use="required"/>
                <xsd:attribute name="to" type="xsd:IDREF" use="required"/>
            </xsd:complexType>
        </xsd:element>
    </xsd:choice>
</xsd:group>
```

Figure A-1. Continued

```
<xsd:group name="queuing">
    <xsd:sequence>
        <xsd:element name="queuing">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:group ref="queuingNodes" minOccurs="0" maxOccurs="unbounded"/>
                    <xsd:group ref="queuingEdges" minOccurs="0" maxOccurs="unbounded"/>
                </xsd:sequence>
                <xsd:attribute name="id" type="xsd:ID" use="required"/>
                <xsd:attribute name="src" type="xsd:anyURI" use="optional"/>
            </xsd:complexType>
        </xsd:element>
    </xsd:sequence>
</xsd:group>
<!-- added by nklim -->
<xsd:group name="queuingNodes">
    <xsd:choice>
        <xsd:group ref="facility"/>
        <xsd:group ref="source"/>
        <xsd:group ref="decision"/>
        <xsd:group ref="fork"/>
        <xsd:group ref="joins"/>
        <xsd:group ref="sink"/>
        <xsd:group ref="block"/>
    </xsd:choice>
</xsd:group>
<xsd:group name="qBlock">
    <xsd:choice>
        <xsd:element name="qBlock" minOccurs="0" maxOccurs="unbounded">
            <xsd:complexType>
                <xsd:choice>
                    <xsd:group ref="models"/>
                </xsd:choice>
                <xsd:attribute name="id" type="xsd:ID" use="required"/>
            </xsd:complexType>
        </xsd:element>
    </xsd:choice>
</xsd:group>
```

Figure A-1. Continued

```
    <xsd:group name="block">
        <xsd:choice>
            <xsd:element name="block" minOccurs="0" maxOccurs="unbounded">
                <xsd:complexType>
                    <xsd:choice>
                        <xsd:group ref="models"/>
                    </xsd:choice>
                    <xsd:attribute name="id" type="xsd:ID" use="required"/>
                    <!-- xsd:attribute name="qSize" type="xsd:Integer" use="required"/-->
                    <!--for Queuing Size-->
                    <!--xsd:element name ="blockingInput" type="boolean" default="false"
minOccurs="1"/-->
                </xsd:complexType>
            </xsd:element>
        </xsd:choice>
    </xsd:group>
    <xsd:group name="facility">
        <xsd:choice>
            <xsd:element name="facility" minOccurs="0" maxOccurs="unbounded">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:group ref="handles"/>
                        <xsd:group ref="queue"/>
                        <xsd:group ref="server"/>
                        <xsd:group ref="queuingEdges"/>
                    </xsd:sequence>
                    <xsd:attribute name="id" type="xsd:ID" use="required"/>
                    <!--xsd:attribute name="numberOfserver" type="xsd:nonNegativeInteger"
use="required"/-->
                </xsd:complexType>
                <!--xsd:element name ="blockingInput" type="boolean" default="false"
minOccurs="1"/-->
            </xsd:element>
        </xsd:choice>
    </xsd:group>
    <xsd:group name="decision">
        <xsd:choice>
            <xsd:element name="decision" minOccurs="0" maxOccurs="unbounded">
                <xsd:complexType>
```

Figure A-1. Continued

```
                    <xsd:sequence>
                        <xsd:group ref="handles"/>
                        <xsd:group ref="scripts"/>
                    </xsd:sequence>
                    <xsd:attribute name="id" type="xsd:ID" use="required"/>
                </xsd:complexType>
            </xsd:element>
        </xsd:choice>
    </xsd:group>
    <xsd:group name="joins">
        <xsd:choice>
            <xsd:element name="join" minOccurs="0" maxOccurs="unbounded">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:choice>
                            <xsd:group ref="handles"/>
                            <xsd:group ref="scripts"/>
                        </xsd:choice>
                    </xsd:sequence>
                    <xsd:attribute name="id" type="xsd:ID" use="required"/>
                    <xsd:attribute name="collectionSize" type="xsd:integer" default="1"/>
                </xsd:complexType>
                <!--xsd:element name ="blockingInput" type="boolean" default="false"
minOccurs="1"/-->
            </xsd:element>
        </xsd:choice>
    </xsd:group>
    <xsd:group name="fork">
        <xsd:choice>
            <xsd:element name="fork" minOccurs="0" maxOccurs="unbounded">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:group ref="handles"/>
                        <xsd:group ref="distributions"/>
                    </xsd:sequence>
                    <xsd:attribute name="id" type="xsd:ID" use="required"/>
                </xsd:complexType>
            </xsd:element>
        </xsd:choice>
```

Figure A-1. Continued

```
    </xsd:group>
    <xsd:group name="distributions">
        <xsd:choice>
            <xsd:element name="distribution" minOccurs="0" maxOccurs="unbounded">
                <xsd:complexType>
                    <xsd:attribute name="id" type="xsd:ID" use="required"/>
                    <xsd:attribute name="distributionFunction" type="xsd:string" default="uniform"/>
                    <xsd:attribute name="parameter_lowerBound" type="xsd:unsignedInt"
default="0"/>
                    <xsd:attribute name="parameter_upperBound" type="xsd:unsignedInt"
default="0"/>
                </xsd:complexType>
            </xsd:element>
        </xsd:choice>
    </xsd:group>
    <xsd:group name="sink">
        <xsd:choice>
            <xsd:element name="sink" minOccurs="0" maxOccurs="unbounded">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:group ref="handles"/>
                        <xsd:group ref="scripts"/>
                    </xsd:sequence>
                    <xsd:attribute name="id" type="xsd:ID" use="required"/>
                </xsd:complexType>
                <!--xsd:element name ="blockingInput" type="boolean" default="false"
minOccurs="1"/-->
            </xsd:element>
        </xsd:choice>
    </xsd:group>
    <xsd:group name="source">
        <xsd:choice>
            <xsd:element name="source" minOccurs="0" maxOccurs="unbounded">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:group ref="entities"/>
                        <xsd:group ref="handles"/>
                        <xsd:group ref="scripts"/>
                    </xsd:sequence>
```

Figure A-1. Continued

```
                    <xsd:attribute name="id" type="xsd:ID" use="required"/>
                </xsd:complexType>
            </xsd:element>
        </xsd:choice>
    </xsd:group>
    <xsd:group name="entities">
        <xsd:choice>
            <xsd:element name="entity" minOccurs="0" maxOccurs="unbounded">
                <xsd:complexType>
                    <xsd:attribute name="id" type="xsd:ID" use="required"/>
                    <xsd:attribute name="generationFunction" type="xsd:string" default="uniform"/>
                    <xsd:attribute name="parameter_1" type="xsd:unsignedInt" default="1"/>
                    <xsd:attribute name="parameter_2" type="xsd:unsignedInt" default="1"/>
                </xsd:complexType>
            </xsd:element>
        </xsd:choice>
    </xsd:group>
    <xsd:group name="queue">
        <xsd:choice>
            <xsd:element name="queue" minOccurs="0" maxOccurs="unbounded">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:group ref="handles"/>
                        <xsd:group ref="scripts"/>
                    </xsd:sequence>
                    <xsd:attribute name="id" type="xsd:ID" use="required"/>
                    <xsd:attribute name="qSize" type="xsd:nonNegativeInteger" use="optional"/>
                    <xsd:attribute name="serviceTime" type="xsd:unsignedLong" use="required"/>
                </xsd:complexType>
            </xsd:element>
        </xsd:choice>
    </xsd:group>
    <xsd:group name="server">
        <xsd:choice>
            <xsd:element name="server" minOccurs="0" maxOccurs="unbounded">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:group ref="handles"/>
                        <xsd:group ref="scripts"/>
```

Figure A-1. Continued

```
                    </xsd:sequence>
                    <xsd:attribute name="id" type="xsd:ID" use="required"/>
                </xsd:complexType>
            </xsd:element>
        </xsd:choice>
    </xsd:group>
    <xsd:group name="queuingEdges">
        <xsd:choice>
            <xsd:element name="trace" minOccurs="0" maxOccurs="unbounded">
                <xsd:complexType>
                    <xsd:attribute name="from" type="xsd:IDREF" use="required"/>
                    <xsd:attribute name="to" type="xsd:IDREF" use="required"/>
                </xsd:complexType>
            </xsd:element>
        </xsd:choice>
    </xsd:group>
</xsd:schema>
```

Figure A-1. Continued

LIST OF REFERENCES

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

[2] Hall, R. R. Queuing Methods. Englewood Cliffs, NJ: Prentice-Hall. 1991.

[3] Mahoney, R. Computer Simulation: A Practical Perspective. San Diego, CA: Academic Press. 1991.

[4] Kheir N.A. Systems Modeling and Computer Simulation. New York, NY: Dekker. 1996.

[5] World Wide Web Consortium (W3C). 2004. eXtensible Markup Language (XML). Available from URL: http://www.w3.org/TR/REC-xml. Site last visited May 2004.

[6] Fishwick, P.A. Using XML for Simulation Modeling. Proceedings of the Winter Simulation Conference. Orlando, FL. 2002

[7] World Wide Web Consortium (W3C). 2001. XML Schema (XML). Available from URL: http://www.w3.org/TR/xmlschema-0/. Site last visited May 2004.

[8] Walsh, N. Understanding XML Schemas. 1999. Available from URL: http://www.xml.com/pub/a/1999/07/schemas/index.html?page=3. Site last visited May 2004.

[9] World Wide Web Consortium (W3C). 2003. The eXtensible Stylesheet Language (XSL). Available from URL: http://www.w3.org/Style/XSL. Site last visited May 2004.

[10] Kay, M. XSLT. West Sussex, Canada: Wrox. 2002.

[11] The Apache Software Foundation. 2004. Xalan-Java. Available from URL: http://xml.apache.org/xalan-j/index.html. Site last visited May 2004.

[12] Kim, T., Lee, J., Fishwick, P.A. A Two-Stage Modeling and Simulation Process for Web-Based Modeling and Simulation. ACM Transactions on Modeling and Computer Simulation (TOMACS). Vol.12, No.3, pp 230-248. 2002

[13] Shim, H. An XML-based Diagrammatic Dynamic Modeling And Simulation System [dissertation]. Gainesville: University of Florida. 2003.

[14] Hopkins, J.F. An Investigation of the Use of Metaphor in the *rube*TM Paradigm [dissertation]. Gainesville: University of Florida. 2001.

[15] Veeke, H.P.M., Ottjes, J.A. TOMAS: Tool for Object-Oriented Modeling and Simulation. Proceedings of the Business and Industry Simulation Symposium (ASTC 2000). Available from URL http://www.wbmt.tudelft.nl/tt/users/duinkerk/papers/was0004b.pdf. Site last visited May 2004.

[16] TOMASWEB. 2003. Modeling and Simulation for Discrete Processes in Production and Transportation. Available from URL http://www.tomasweb.com/index2.html. Site last visited May 2004.

[17] Vangheluwe, H., Lara, J.D., Meta-Models are Models Too. Proceedings of Winter Simulation Conference. Orlando, FL. 2002.

[18] Modelling, Simulation and Design Lab, 2002, ATOM3. Available from URL:http://atom3.cs.mcgill.ca/index_html. Site last visited May 2004.

[19] World Wide Web Consortium (W3C). 2004. Simulation Reference Markup Language (XML). Available from URL: http://www.w3.org/TR/SRML. Site last visited May 2004.

[20] Reichenthal, S.W. The Simulation Reference Markup Language (SRML): A Foundation for Representing BOMs and Supporting Reuse. Available from URL: http://www.simventions.com/whitepapers/02F-SIW-038.pdf. Site last visited May 2004.

[21] The JavaTM Tutorial, Creating a GUI with JFC/Swing, Available from URL: http://java.sun.com/docs/books/tutorial/uiswing. Site last visited May 2004.

[22] Computing-Laboratory, the JavaSim User's Manual, Department of Computing Science, University of Newcastle upon Tyne. 1999.

[23] Arief, L.B., Speirs, N.A. Automatic Generation of Distributed System Simulations from UML. Proceedings of the13th European Simulation Multiconference (ESM'99). Warsaw.1999.

[24] Park, M. SimPackJ/S: A Web-Oriented Toolkit for Discrete Event Simulation, in Enabling Technologies for Simulation Science [dissertation]. Gainesville: University of Florida. 2002.

BIOGRAPHICAL SKETCH

NamKyu Lim was born May 19, 1971, in Kwangju, Republic of Korea. He received his Bachelor of Science degree in computer science from the Air Force Academy, Republic of Korea, in May 1994. In 2004, he will receive his Master of Science degree in computer and information science and engineering at the University of Florida. His major research area is computer modeling and simulation