ARCHITECTURE FOR A LOW-LEVEL FUNCTIONAL SPECIFICATION
LANGUAGE SUPPORTING MULTIMODELING AND SIMULATION

By

JINHO LEE

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2005
To my parents, sister, and brother
ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my advisor, Dr. Paul A. Fishwick, who gave me inspiration and guidance throughout my Ph.D. studies at the University of Florida. I would also like to give my sincere appreciation to my supervisory committee members (Dr. Beverly Sanders, Dr. Abdelsalam A. Helal, Dr. Shigan Chen, and Dr. Sherman Bai) for their precious time and advice.

I appreciate all the colleagues in our research group for sharing valuable ideas. Special thanks go to Minho Park and Hyunju Shim for their help and companionship.

Also, I am grateful to the National Science Foundation and the Air Force Research Laboratory for their financial support for my studies in the United States.

I owe great love to my parents, who prayed and encouraged me throughout my studies; and to my brother and sister. They are the reason for my existence.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGMENTS</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
</tr>
<tr>
<td>ABSTRACT</td>
</tr>
<tr>
<td>CHAPTER</td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
</tr>
<tr>
<td>INTRODUCTION</td>
</tr>
<tr>
<td>1.1 Motivations and Challenges</td>
</tr>
<tr>
<td>1.2 Contributions to Knowledge</td>
</tr>
<tr>
<td>1.2.1. Web-Based Modeling and Simulation by Using XML to Represent Simulation Models</td>
</tr>
<tr>
<td>1.2.2 Multimodeling Using MXL, DXL, and an MXL-DXL Bridge</td>
</tr>
<tr>
<td>1.2.3 Flexible Diagrammatic Presentation of Functionality for DXL Blocks</td>
</tr>
<tr>
<td>1.2.4 Ports-based Modeling and Simulation Capable of Encoding Documents Rather Than Data</td>
</tr>
<tr>
<td>1.2.5 Explicit Support for Discrete Event and Continuous Simulation</td>
</tr>
<tr>
<td>2 BACKGROUND</td>
</tr>
<tr>
<td>BACKGROUND</td>
</tr>
<tr>
<td>2.1 General Multimodels</td>
</tr>
<tr>
<td>2.1.1 Intra-level Coupling and Inter-level of Multimodels</td>
</tr>
<tr>
<td>2.1.2 Homogeneous and Heterogeneous Multimodels</td>
</tr>
<tr>
<td>2.2 The RUBE Framework</td>
</tr>
<tr>
<td>2.3 SimPack Tools for the RUBE Framework</td>
</tr>
<tr>
<td>2.3.1 Event Scheduling Methods</td>
</tr>
<tr>
<td>2.3.2 Resource Allocation</td>
</tr>
<tr>
<td>2.3.3 Simulation State and Tracing</td>
</tr>
<tr>
<td>2.3.4 Random Number Generators</td>
</tr>
<tr>
<td>3 A LOW-LEVEL FUNCTIONAL SPECIFICATION LANGUAGE FOR MULTIMODELING AND SIMULATION</td>
</tr>
<tr>
<td>A LOW-LEVEL FUNCTIONAL SPECIFICATION LANGUAGE FOR MULTIMODELING AND SIMULATION</td>
</tr>
<tr>
<td>3.1 The Concept of Dynamic eXchange Language (DXL)</td>
</tr>
</tbody>
</table>
6.1.5 Explicit Support for Discrete Event and Continuous Simulation...........66
6.2 Future Research ........................................................................................................66
   6.2.1 Support of Different Model Types for Multimodeling.................................66
   6.2.2 Distributed Simulation......................................................................................66

APPENDIX

SIMPACK FOR PYTHON .............................................................................................69
LIST OF REFERENCES .................................................................................................87
BIOGRAPHICAL SKETCH ............................................................................................93
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>General Multimodel Structure</td>
<td>10</td>
</tr>
<tr>
<td>2-2</td>
<td>Homogeneous Refinement: declarative → declarative</td>
<td>11</td>
</tr>
<tr>
<td>2-3</td>
<td>Homogeneous Refinement: functional → functional</td>
<td>12</td>
</tr>
<tr>
<td>2-4</td>
<td>Heterogeneous Refinement: functional → declarative</td>
<td>12</td>
</tr>
<tr>
<td>2-5</td>
<td>Heterogeneous Refinement: declarative → functional</td>
<td>13</td>
</tr>
<tr>
<td>2-6</td>
<td>The RUBE Framework</td>
<td>14</td>
</tr>
<tr>
<td>3-1</td>
<td>The Visual Syntax of DXL</td>
<td>20</td>
</tr>
<tr>
<td>3-2</td>
<td>The Visual Syntax for a multimodel of DXL</td>
<td>21</td>
</tr>
<tr>
<td>3-3</td>
<td>The Schema of DXL Element</td>
<td>22</td>
</tr>
<tr>
<td>3-4</td>
<td>The Schema of Block Element</td>
<td>23</td>
</tr>
<tr>
<td>3-5</td>
<td>The Schema of Port Element</td>
<td>24</td>
</tr>
<tr>
<td>3-6</td>
<td>The Schema of Definition Element</td>
<td>25</td>
</tr>
<tr>
<td>3-7</td>
<td>The Schema of Connect Element</td>
<td>26</td>
</tr>
<tr>
<td>3-8</td>
<td>The Schema of Simulation Element</td>
<td>27</td>
</tr>
<tr>
<td>3-9</td>
<td>Block Notation of DXL</td>
<td>28</td>
</tr>
<tr>
<td>3-10</td>
<td>Diagrammatic Presentation of DXL</td>
<td>29</td>
</tr>
<tr>
<td>3-11</td>
<td>Model of DXL for Figure 4-2 Example</td>
<td>30</td>
</tr>
<tr>
<td>3-12</td>
<td>Examples of SEND Pseudo Code</td>
<td>31</td>
</tr>
<tr>
<td>3-13</td>
<td>The XML-based Dataflow in DXL</td>
<td>32</td>
</tr>
<tr>
<td>3-14</td>
<td>Example of Synchronous Inputs</td>
<td>33</td>
</tr>
</tbody>
</table>
Abstract of Dissertation Presented to the Graduate School of the University of Florida in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

ARCHITECTURE FOR A LOW-LEVEL FUNCTIONAL SPECIFICATION LANGUAGE SUPPORTING MULTIMODELING AND SIMULATION

By

Jinho Lee

August, 2005

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

Multimodels are models composed of heterogeneous dynamic system components. Since most normal models like FSM and FBM represent only a part of the overall system behavior, they make only a subset of solution for analyzing the prediction and diagnosis of our real world. Multimodels can, on the other hand, give abstraction perspectives for a complex real-world system. There is a need for a unified, low-level functional specification language (and associated diagrammatic presentation) for simulations that may be expressed in a wide variety of modeling visualizations and formalisms.

Our objective was to design a low-level functional specification language, with a circuit-like appearance and functionality, to more efficiently support eXtensible Markup Language (XML)-based multimodeling and simulation. For the multimodeling and simulation, there are three stages in our approach.

For the first stage, we have defined traditional heterogeneous models using the Multimodeling eXchange Language (MXL). These models are transformed into a low-
level functional specification language through eXtensible Stylesheet Language Transformation (XSLT) translator.

For the second stage, we have designed a low-level functional specification language and its diagrammatic presentation for multimodeling and simulation, the Dynamic eXchange Language (DXL). DXL is an XML-based language that positions itself between higher-level modeling languages and a programming code.

For the last stage, by using the XML Document Object Model (DOM), we demonstrate a translation approach that yields a target code for simulation such as JavaScript and Python. Through these three stages, we have developed the multimodeling methodology through the low-level functional language and its translation from MXL to simulation codes.
CHAPTER 1
INTRODUCTION

1.1 Motivations and Challenges

Computer modeling and simulation fields have been applied to analyze and expect a variety of complex real-world problems, spreading over academia, the military, and industry. They have tried to satisfy all the requirements of modeling and simulation from the representation to the execution procedure to reasonably simulate and analyze a complex system. But, the complexity of a real-world system has made it difficult to develop a methodology for the specification and analysis of the system. Most of the researchers have specified and analyzed the system having only a simplified and different viewpoint [28, 34, 68].

On the other hand, Web-based modeling and simulation [22, 48, 49] is one of many paradigms for the modeling and simulation. Many problems must be solved for the paradigm to prosper. An XML [6] -based modeling and simulation methodology can satisfy requirements of the Web-based modeling and simulation as well as the traditional modeling and simulation.

Therefore, a unified XML-based methodology is need to support all the properties of those modeling methodologies and simulation techniques to specify and analyze the real-world system. An objective was to design a unified, functional specification language and associated diagrammatic presentation to allow simulations to be expressed in a wide variety of modeling visualizations and formalisms.
Among many existing methodologies, an XML-based functional block diagram methodology might play an important role in accelerating the growth of the fields and satisfying the above requirements. Our two main approaches were the decomposition and the integration of existing modeling methodologies based on an XML-based functional block methodology. We designed a unified low-level functional specification language based on XML for this purpose.

Decomposition has two steps: defines its atomized elements to specify a model and its behaviors using XML, translate each heterogeneous traditional model into homogeneous models consisting of the atomized elements.

Generally, because it is easier to integrate homogeneous components than heterogeneous components of models, sub-models of heterogeneous components are integrated after they are translated into sub-models of homogeneous components. Uniting the notations also helps to reuse model components.

This flexibility and reusability led to creation of two XML-based languages for the RUBE framework. The low-level functional specification language we used is called the DXL [37, 38]; and it includes a higher-level dynamic model wrapper language, the MXL [31, 32, 33].

MXL is an XML-based modeling language to support traditional heterogeneous model types such as Finite State Model (FSM), Functional Block Model (FBM), or Petri net [24, 25, 26, 30, 31]. DXL is an XML-based simple functional block language to support actual simulated execution on the RUBE framework.
The independence between traditional abstract model types and the contemporary actual simulation programs makes it easy to provide model composability and reusability, which are key characteristics for multimodeling.

1.2 Contributions to Knowledge

1.2.1. Web-Based Modeling and Simulation by Using XML to Represent Simulation Models

Web-based modeling and simulation have many advantages. Models can be reviewed with clients on the Internet during model development. Developers can make changes to a model on the Internet without traveling to a client site. The model can access actual model files and data files from any location in the client network [20, 65].

Both MXL and DXL use XML to support model composability and reusability for Web-based modeling and simulation. Heterogeneous traditional models use MXL, and homogeneous primitive models use DXL. Each XML-based model might include its own type of zero or more XML-based models as well as its own atomic XML elements. The MXL might support heterogeneous multimodeling and DXL might support homogeneous multimodeling through the hierarchy of the models. Because both languages use XML, it is easy to transform a heterogeneous model into a homogeneous model using XSLT.

Simulation Reference Markup Language (SRML) [57] is an XML-based modeling and simulation language for a discrete event simulation. SRML is an XML application that can describe the behavior of distributed simulation models, and its runtime environment is software capable of executing those models based on script codes defined in its documents. It defines specific simulation, item, and script elements and their attributes, to support a centralized discrete event simulation. But SRML is a language for
a traditional discrete event simulation. So, it has some limitations to represent the multimodels providing synthetically a variety of viewpoints for a system.

Extensible Modeling and Simulation Framework (XMSF) [8, 9] is a composable set of standards, profiles, and recommended practices for Web-based modeling and simulation. It integrates XML, Internet technologies, and Web Services for supporting its modeling and simulation framework. Its members work for three representative fields which are Web and XML technologies, Internet and networking, and modeling and simulation fields. Their research focuses on Web-based Distributed Interactive Simulation (DIS).

The Simulation Modeling Language (SML) is an open source, web-based, and multi-language simulation project guided by a consortium of industrial, academic, and government simulation consultants and developers [29]. The objective of SML is to produce reusable simulation software at both the simulation source code and modeling source-code levels. The properties of SML are readability, modularity, and extendibility and it supports Java-based, any object-oriented language using design patterns like the common object-oriented, process-oriented modeling design pattern or the entity-object-thread design pattern.

1.2.2 Multimodeling Using MXL, DXL, and an MXL-DXL Bridge

Our approach supports multimodeling using an MXL-DXL bridge where heterogeneous multimodels are specified with MXL and then translated into homogeneous multimodels using a low-level functional specification language for multimodeling and simulation (DXL). Multimodeling is an extension of the more traditional combined modeling techniques [47, 65] which normally combine discrete event and continuous model formalisms in the same model. We approached
multimodeling transforming heterogeneous models into homogeneous models, and approached functional composition with coupled ports of the same data type.

The approach to multimodeling and simulation is interleaving design and simulation. The design process consists of decomposition and composition. High-level functions are hierarchically decomposed into functions for subsystems like intra-level coupling of multimodeling. These subfunctions are then mapped to physical components [50, 62, 63]. During composition, interactions among components are defined. These are similar to inter-level coupling of multimodeling.

PtolemyII [10, 41] provides heterogeneous, concurrent modeling and design for an embedded system using MoML, which is an internal language for PtolemyII. MoML [36] is an XML-based functional block modeling language and supports multimodeling for PtolemyII. This project focuses on inter-level coupling of heterogeneous and homogeneous models like MXL in the RUBE framework [30, 31, 32, 33]. But they try to compose heterogeneous models directly to support multimodeling unlike our MXL-DXL bridge. DXL supports a multimodeling methodology [20, 46, 72] using simple homogeneous blocks, unlike the more complex blocks of MoML.

There are many object-oriented languages for discrete event systems [71, 72] and continuous systems [2, 16, 17, 23, 55]. An object-oriented model derived from a parent model inherits the parent’s interface and equations. The child model can be extended by including additional interactions through the interface and implementation of additional equations [16, 17, 55, 71]. Designing an object-oriented model produces a hierarchical organization of models, and supports reuse and extension of simulation models.
1.2.3 Flexible Diagrammatic Presentation of Functionality for DXL Blocks

In DXL, models are specified by connecting inputs and outputs of primitive models such as multipliers or adders. Each block can be encoded in either JavaScript or Python. Multiple programming languages such as JavaScript and Python [7] become target codes for DXL-to-Simulation translators and an actual data flow for simulation is achieved if we use XML data and schema. DXL supports the combination of lower-level models of subsystems for complex systems hierarchically, which translates to homogeneous multimodeling.

We created a visual formalism for DXL and MXL that operates like a digital circuit. The circuits are fairly flexible, allowing for the three specified modes of operation: sequential, distributed, and hardware. While most work has supported sequential operation, distributed and hardware modes are forthcoming. For sequential simulation, the simulation models described by DXL are translated into programming languages (such as Java, JavaScript, and Python). This DXL-to-Simulation translation approach supports information streaming that operates according to the different modes. For example, message passing can result in future-event list scheduling (for sequential architectures), actual messages (distributed network), or communications busses (hardware). Distributed simulation mode might be based on XML-based message passing protocol, Simple Object Access Protocol (SOAP).

Bond graph modeling [54, 60] defines a system in terms of a group of elements with bonds connecting each element to the rest of the network, and is based on energy-conserving junctions that connect energy storing or transforming elements through bonds. Even though the domains, such as mechanical, electrical, and thermal, are different, each domain has an analogous component in every other domain such as power and energy.
But, bond graphs are inconvenient for the 3D mechanics [67] modeling or continuous-discrete hybrid systems [14, 18].

Linear graphs represent the energy flow through the system component expressed using terminal variables like Kirchhoff’s voltage and current laws. An edge shows an energy flow in a system using an equation expressing relations between terminal variables, and the terminals of components are the nodes of the graph. Linear graphs are more convenient for the 3D mechanics [3, 43] or hybrid systems [45] because of their domain independence and direct representation of the system topology.

Architecture Description Languages (ADLs) [13, 35, 44] are modeling notations for architecture-based development of computer software. Many ADLs for specific domain and general purpose in the software engineering field have been proposed. Though their main focus is an architecture-based design methodology of software, they use similar terminologies as DXL components and generate actual programming code. But most of the ADLs are not XML-based except for a few languages [13]. The XML-based ADL provides a mechanism for rapidly developing new ADLs and the reusability of software components like DXL provides a mechanism for developing and reusing actual simulation codes to analyze a system. ACME [35], which is called an interchange language, supports integration of different ADLs and proposes a formal semantic to facilitate architecture connection [1].

1.2.4 Ports-based Modeling and Simulation Capable of Encoding Documents Rather Than Data

Ports constitute the interface that defines the boundary of components or subsystems in a system configuration. MXL and DXL ports are used for port coupling, where all ports match in number and data type. MXL ports are ports for conceptual
multimodeling in rube framework and DXL ports maintain the conceptual multimodeling and support the multimodel execution. Because of the XML-based environment, DXL ports are capable of encoding documents rather than data, or more specifically XML documents. Thus, a DXL or MXL model supports XML “information streaming” and not only streaming of simple data types. The “types” are accompanied by XML schemata that define the typing structure.

Physical-based modeling and simulation researchers adopt port-based modeling and simulation [16, 17, 39, 42, 66]. They use the ports as an interface for components. Their ports express relations like an energy flow between physical objects like mathematical equations. Port-based product models can be used to generate multiple simulation models at different levels of abstraction. Mechatronics or complex hybrid systems could use port-based modeling and simulation efficiently.

1.2.5 Explicit Support for Discrete Event and Continuous Simulation

For high-fidelity simulation of multidisciplinary systems, hybrid modeling and simulation is required to support a combination of continuous time physical behavior and events occurring at discrete time and space. Because mechatronic systems combine both continuous time phenomena and discrete time events, they require mixed continuous-discrete models [4, 24, 72]. DXL explicitly supports simulation in terms of the development of commonly used simulation blocks found in both discrete-event and continuous simulation.

Modelica [16, 17, 42, 66] is an object-oriented modeling language using the approach for complex and heterogeneous physical systems. It has been developed primarily by the continuous systems community. Modelica provides a 2D-based design in a high-level abstraction, as well as a detailed model execution by using equations for
specifying interrelation of components for the physical system. Modelica has the advantage that is object-oriented with support for model inheritance and sub-typing. It also supports the definition of aggregate connections and signal variables from different energy domains can be combined. It is possible to reuse many physical-based components and simulation analysis tools. But this approach has some limitations to represent a general model using different model types, since it is based on a physical system.

Very High-Speed Integrated Circuit Hardware Description Language—Analog and Mixed-Signal (VHDL-AMS) is an extension from VHDL to support Analog and Mixed-Signal components. VHDL-AMS has more modeling primitives for discrete-event simulation than continuous time simulation. VHDL is limited to encapsulation but has been standardized by IEEE.
CHAPTER 2
BACKGROUND

2.1 General Multimodels

2.1.1 Intra-level Coupling and Inter-level of Multimodels

Multimodels are models that are composed of heterogeneous models. Since most normal models like FSM and FBM represent only a part of the overall system behavior, they make only a subset of a solution for analyzing the prediction and diagnosis of our real world. Multimodels can, on the other hand, get a number of abstraction perspectives for a complex real-world system. Therefore, they can more correctly represent and analyze a complex real-world system. Different component models in a multimodel can define the activity at different stages of the simulation.

Figure 2-1. General Multimodel Structure

Figure 2-1 shows a multimodel example having an abstraction hierarchy of heterogeneous models. Double dotted lines in this figure mean the relation of its components to compose an upper level model. This relation should ensure intra-level coupling. Intra-level coupling defines model components coupled to one another in the
same model. In Figure 2-1, there are two intra-level couplings. $M_2$ is composed of $m_1$, $m_2$, and $m_3$. $M_3$ is composed of $s_1$, $s_2$, and $s_3$. The intra-level coupling in $M_2$ defines how the sub-models of $M_2$ are formed to represent a model $M_2$ and the intra-level coupling in $M_3$ defines how the sub-models of $M_3$ are formed to represent a model $M_3$.

![Diagram](image)

**Figure 2-2. Homogeneous Refinement: declarative → declarative**

In Figure 2-1, down-arrows mean refining a model and up-arrows mean the abstraction of a model. These two relations should ensure inter-level coupling. Inter-level coupling defines rules as to how model components from one model can be refined into models of different types. In Figure 2-11, there are two inter-level couplings. $M_2$ refines a model $M_1$ and $M_3$ refines $m_2$. A model $m_2$ abstracts $M_3$ and $M_1$ abstracts $M_2$. To refine $m_2$ into $M_3$, we should ensure the intra-level coupling in $M_2$ when we change the sub-model $m_2$ and $M_3$.

### 2.1.2 Homogeneous and Heterogeneous Multimodels

In Figure 2-2, the Finite State Automata (FSA) shows top-down homogeneous decompositions which means intra-level coupling because the element of the upper model is defined by the same type of models. State $s_1$ is decomposed into the lower level of FSA. The predicates $p_1(i)$ and $p_2(i)$ involves external input variable $i$ such as $i=0$. 
Figure 2-3. Homogeneous Refinement: functional $\rightarrow$ functional

Figure 2-3 shows a two-level functional hierarchy where function $f$ is defined in terms of a composition of three other functions $f_1, f_2,$ and $f_3.$

Figure 2-4. Heterogeneous Refinement: functional $\rightarrow$ declarative

Heterogeneous decomposition of models in intra-level coupling describes the semantics of a model using different models semantics. The model of Figure 2-4 shows FSA that defines the internal state transitions associated with function $f.$ The predicate $p(i)$ tests input variable $i.$ All FSAs should have this kind of functional box around them to represent explicit external input and output semantics even though the functional box is not always explicit in the FSA graphical representation.

Figure 2-5 shows a state-to-state space mapping, which is not immediately apparent from the figure. Specifically, most functional block models that represent some aspect of
physical reality involve state transitions which means \( f_1 \) and \( f_2 \) contain internal state transitions. There is a transition with two possible types of semantics coming out of state \( s \) in Figure 2-5. An external transition would be of the form \( p_2(i) \) and an internal transition would be based on a variable \( o \) that is a component of internal state space of \( f_1 \) and \( f_2 \).

![Figure 2-5. Heterogeneous Refinement: declarative → functional](image)

### 2.2 The RUBE Framework

The RUBE framework is an XML-based dynamic modeling and simulation framework permitting the users to specify and execute a dynamic model, with an ability to customize a model presentation using 2D or 3D visualization [11, 12, 26, 40, 52, 61]. The purpose of RUBE is to facilitate a dynamic multimodel construction based on XML, and visualize and execute the model within a 2D environment [52] or a 3D immersive environment [53]. The overall process of the XML-based RUBE framework is shown in Figure 2-6.

In Figure 2-6, the entire process of the RUBE framework is described by starting independently two flows of methodologies which are model representation and model
creation, and then merging the flows and creating final model visualization and execution codes.

Figure 2-6. The RUBE Framework

For model representation, a dynamic model is a scene file, which contains 2D or 3D geometry objects. The RUBE framework has a 2D or a 3D interface using *Sodipodi* [64] and *Blender* [5, 58, 59] for each representation of the 2D or the 3D model scenes. The scene files just represent the appearance of geometric objects in the model and do not have any information about model behavior or dynamics. The scene files can be either standard 2D or 3D XML documents which are specified by Scalar Vector Graphics (SVG) [15] or eXtensible 3D (X3D) [69] respectively. Therefore, any 2D or 3D tools which can generate SVG or X3D files might be applied as a part of the RUBE framework.

For model creation, there are two stages: model translation and model simulation. For model translation, a dynamic model is an actual model file which is represented by MXL. The MXL file describes the behavior of the model and represents the model file that includes the specification of a heterogeneous multimodel in an abstract level such as FBM (Functional Block Model) and FSM (Finite State Machine). The MXL-to-DXL translator using eXtensible Stylesheet Language Transformation (XSLT) [27] translates a
model file written in MXL into a low-level functional specification language called DXL, which can be described with a homogeneous block diagrammatic presentation. For model simulation, the DXL is translated into an executable programming code for the model simulation using the DXL-to-Simulation translator. The programming code can be executed using SimpackJ/S or Simpack Python [19, 51], which provides the underlying code foundation for libraries, classes, and objects for simulation.

For model merging, a 2D or a 3D merge engine, which are created with XSLT, merges two XML documents: a scene file and the actual model execution file. And then, the engine generates the final RUBE visualization and execution file of the model. The file might visualize and execute the model within a 2D environment, Sodipodi, or a 3D immersive environment, Blender. The current RUBE framework also supports integrative multimodeing providing a human-computer interaction environment that allows components of different model types to be linked to one another [53].

The red line rectangle in Figure 2-6 shows my contributions in the RUBE Framework. They are the design of the output DXL of the MXL-to-DXL translator, the design of DXL, DXL-to-Simulation translator, and the final simulation code.

### 2.3 SimPack Tools for the RUBE Framework

SimPack is a software toolkit for event scheduling and queuing model simulations. It is a collection of multiple programming languages tools for computer simulation such as C, C++, Java, JavaScript, and Python. The RUBE framework especially uses Java, JavaScript, and Python to support Web-based modeling visualization and simulation as well as traditional programming language simulation. The generated simulation codes in the RUBE framework reference SimPack as basic simulation APIs. SimPack provides four sorts of methods according to the functionality.
2.3.1 Event Scheduling Methods

Five methods are used to provide a means to simulate basic models involving event scheduling by dealing with managing the future event list (FEL).

- **Sim.init**: This method initializes the discrete event simulation, and it is therefore mandatory to call it at the top of the code.

- **Sim.schedule**: This method schedules an event to occur in the future. This inserts the event within the future FEL according to increasing time for each event. The FEL is ordered on simulation time at which the future event will occur, so no matter in what order events are scheduled, or put into the FEL, they come out of the FEL in ascending order of their occurrence times. In other words, events with smaller times are placed toward the head of the list since they will be processed first. Events with the same occurrence time are scheduled FIFO. In the FEL, the time associated with each event is converted to and stored as a simulation time which is the elapsed time since the start of the simulation, measured in arbitrary time units. The FEL times are absolute virtual times when events are to occur in the simulation. FEL items are each composed of 1) time for the event to occur, 2) the event to occur, 3) the token (which stores attribute data and information for this event), and 4) the priority of this token. Higher priority tokens get scheduled before lower priority ones in the FEL.

- **Sim.next_event**: This method causes the occurrence of whatever future event has the minimum occurrence time, taking this event as the next event from the FEL. Whereas Sim.schedule inserts items into the FEL, Sim.next_event removes the item in the head of the list. Within the same priority level, this means the smallest time value for a token. The order of activities for this function is: to check to see if (1) the list is empty, or (2) whether the synchronization with the physical clock and the virtual time is likely to be less than the physical time. If either of these conditions is true, this function returns a value of “-1”. Otherwise, it returns the event (an integer value). If an event is to be returned (i.e., not “-1”), the virtual clock time is set to the time of the event which has been removed from the head of the FEL.

- **Sim.cancel_event**: This method removes the first item from the FEL with an item specified by the event. If this method is successful in finding this event then 1) the event is removed from the FEL, and 2) the corresponding token number is returned. On the other hand, if there is no such item in the FEL, the FEL remains unchanged.

- **Sim.cancel_token**: This method cancels the first item from the FEL with an item specified by the token identifier. If this method finds this token then 1) the event is removed from the FEL, and 2) the corresponding event identifier is returned. On the other hand, if the token cannot be found then the FEL is unchanged. The token is identified only by its first attribute `attr[0].item`. 
2.3.2 Resource Allocation

There are four resource allocation methods that deal with queuing for a resource or server:

- **Sim.create_facility**: This method creates a new facility or resource, which will develop a service queue behind it. A facility is a shared resource consisting of service and queues where tokens vie for service. For example, a facility can be a row of cashiers, a door (for entering and exiting a room), or a machine that accepts raw parts for processing. The service could be check-in at an airport ticket counter. The queue could be the line in which passengers stand and wait until the service is available at the counter. A facility is a container, or collection, of servers (greater than or equal to 1). If there is one server, the entire facility creates a “single server queue”, otherwise, there are multiple server queues where one facility has more than one server. A queue is composed of tokens, with only one queue per facility.

- **Sim.request**: This method requests service from a facility. In other words, this is the mechanism by which an incoming token requests a server in the facility. The return value of this method is FREE or BUSY. FREE means that the facility had an available server for this token and the token has now started receiving service from the facility. BUSY means that the facility had no available server for this token, and the token has been put into the facility’s priority queue to await availability of a server.

- **Sim.preempt**: This method preempts the server which means a facility. An incoming token requests a facility, and will usually preempt (or replace) the token already being serviced. The difference between preemption and a request is that with preemption, a token does not normally wait in the queue at all which means it bypasses the queue completely to obtain immediate service. The mechanisms are as follows. If the incoming token’s priority is lower than or equal to all tokens currently being served in the facility then that token will be queued. But, if the incoming token’s priority is higher, the server token will be preempted and placed back at the head of the facility queue, time-tagged with its remaining time for service.

- **Sim.release**: This method releases the server for the next token waiting in the queue. This is generally performed after a token has obtained service. That token then releases the facility to the next waiting token at the head of the queue. This is done by automatically scheduling the token on the head of the queue if there is any. The automatic scheduling uses Delta time = 0.0 and the type of event that was used by the token that made the facility request (i.e., prior to having to wait).

2.3.3 Simulation State and Tracing

There is one method used to get state of FEL and facilities, or print their contents.
• **Sim.state**: This method gets the simulation state at any point in time (i.e., wherever this method is invoked in the code).

### 2.3.4 Random Number Generators

There are five methods generating random numbers for different probability density functions:

- **Sim.random**: This method is a uniform random number generator.
- **Sim.uniform**: This method is a uniform random number generator for double floating point.
- **Sim.normal**: This method is a normal random number generator for a distribution with mean \( x \) and standard deviations.
- **Sim.expntl**: This method is an exponential random number generator with mean \( x \).
- **Sim.erlang**: This method is an erlang random number generator for a distribution with mean \( x \) and standard deviations.
CHAPTER 3
A LOW-LEVEL FUNCTIONAL SPECIFICATION LANGUAGE FOR
MULTIMODELING AND SIMULATION

3.1 The Concept of Dynamic eXchange Language (DXL)

DXL is a unified, low-level functional specification language, and associated
diagrammatic presentation, for simulation on the RUBE framework [26, 37, 38]. In
simple DXL models, models are specified by connecting inputs and outputs of primitive
models such as multipliers or adders.

In multimodels, the models might be abstracted upper-layer block as sub-layer
models. DXL supports the combination of lower-level models of subsystems for complex
systems hierarchically which means homogeneous multimodeling. Because all DXL
models have its outermost block as a wrapper, the important characteristics for
multimodeling, composability, and reusability can be supported clearly.

When each block plays a role as a leaf node in the structure of a multimodel, it can
be encoded in one of the programming languages such as JavaScript or Python. Multiple
programming languages become the target codes for DXL-to-Simulation translators and
an actual data flow for simulation is achieved if we use XML data and schema.

In this chapter, a diagrammatic presentation and formalism for DXL, which
operates like a circuit, is created. Ports constitute the interface that defines the boundary
of components or sub-systems in a system configuration. MXL and DXL ports are used
for port coupling, where all ports match in number and data type. MXL ports are for
conceptual multimodeling in rube framework and DXL ports are there to maintain the
conceputal multimodeling and support the multimodel execution. Because of XML-based
environment, DXL ports are capable of encoding documents rather than data, or more
specifically XML documents. Thus, a DXL or MXL model supports XML “information
streaming” and not only streaming of simple data types. The “types” are accompanied by
XML schemata that define the typing structure.

3.2 The Syntax of DXL

3.2.1 The Visual Syntax of DXL

The right part of Figure 3.1 shows visual syntax elements of DXL. DXL elements
are blocks, input and output ports, and connects. Generally, a DXL model is composed of
“block” elements and “connect” elements. A DXL block element is composed of
“port” elements and “its actual computation codes”. DXL block elements are
connected by each connect element through input and output port elements. Input
and output port elements can be included in a block element and each port element
is connected to the corresponding output and input port element of other block
elements for its data flow, which each connect element controls.

Figure 3-1. The Visual Syntax of DXL

The left part of Figure 3.1 shows a simple DXL model consisting of two blocks and
one connect. The meaning of this model is that after the computation of “Block 1” is
finished, the output becomes the input of “Block 2”. Therefore, “Block 2” can execute its own computation using the output of “Block 1” as its input.

Figure 3-2. The Visual Syntax for a multimodel of DXL

DXL block element might include another DXL model when it represents a homogeneous multimodel. In Figure 3-2, “Outer Block B” doesn’t have “its actual computation codes” but another DXL model. Therefore, when “Outer Block B” starts, the input of “Outer Block B” becomes the input of “Inner Block a”. On the other hand, when “Outer Block B” finished, the output of “Inner Block b” becomes the output of “Outer Block B”. According to the flow of “connect” elements, after the inner blocks of “Outer Block B” are executed, the output of “Outer Block B” becomes the input of the “Block C” in Figure 3-2.

3.2.2 The XML Syntax of DXL

3.2.2.1 The DXL Element

The DXL element is the root of a multimodeling and simulation. Each DXL element is an individual model definition in a DXL file. The FBM and FSM in MXL are translated into each homogeneous block model of DXL. It contains block and connect elements as children.
Figure 3-3. The Schema of DXL Element

3.2.2.2 The block Element

The block elements are basic components for a model execution. In other words, these block elements become not only schematic elements of a DXL element but also basic units of actual simulation codes used in an event-based scheduling. The block element contains port, block, and definition elements as children elements.
Figure 3-4. The Schema of Block Element

- **id attribute**: The id attribute specifies the name of the block element. The value of id attribute generated by MXL-to-DXL translator has the meaning of a hierarchical structure because the name of the blocks in the higher layer in a multimodel are included.

- **type attribute**: The type attribute specifies the input property of the block element. DXL blocks support two sorts of properties for input, **synchronous** and **asynchronous**.

- **delta attribute**: The delta attribute specifies the delta time which means a delay time for the block. If the value of delta attribute is greater than 0, its output is transferred to its next blocks after the delaying time passes.

- **internal attribute**: The internal attribute specifies the block generated by MXL-to-DXL translator for the special purpose. For example, we need the special blocks, INPUT and OUTPUT DXL blocks, for the declarative model such as FSM.
3.2.2.3 The port Element

```xml
<xsd:simpleType name="TypeName">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Boolean"/>
    <xsd:enumeration value="Integer"/>
    <xsd:enumeration value="Float"/>
    <xsd:enumeration value="Double"/>
    <xsd:enumeration value="String"/>
    <xsd:enumeration value="LIST"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:group name="ports">
  <xsd:sequence>
    <xsd:element name="port" minOccurs="0" maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:attribute name="id" type="xsd:ID" use="required"/>
        <xsd:attribute name="type" type="TypeName" use="required"/>
        <xsd:attribute name="source" type="xsd:string" use="optional"/>
        <xsd:attribute name="target" type="xsd:string" use="optional"/>
        <xsd:attribute name="data_type" type="xsd:string" use="required"/>
        <xsd:attribute name="initial" type="TypeName" use="optional"/>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:group>
```

Figure 3-5. The Schema of Port Element

An element port is included in a block or connect element that is a direct component of a DXL element. It has four attributes such as id, type, source, and target attributes.

- **id attribute**: The id attribute specifies the name of the port element. The value of id attribute generated by MXL-to-DXL translator has the meaning of a hierarchical structure because the name of the blocks in the higher layer in a multimodel are included.

- **type attribute**: The type attribute specifies the type of data value used in the port element. Objects as well as primitive data types are supported because of the characteristics of script and object-oriented programming language. If simulation codes generated by a DXL model use the APIs creating XML documents into programming objects, XML documents can be
considered as the same objects as other programming objects in this DXL model.

- **source attribute**: The *source* attribute specifies the attribute *id* of the output *port* element that transfer data to current *port* element. In a single layer DXL model, the value of the *source* attribute is the attribute *id* of the output *port* element of other *block* elements, but it can be the attribute *id* of the input *port* element of its own outer *block* element in a DXL multimodel.

- **target attribute**: The *target* attribute specifies the attribute *id* of the input *port* element to which current *port* element transfer data. In a single layer DXL model, the value of the *target* attribute is the attribute *id* of the input *port* element of other *block* elements, but it can be the attribute *id* of the output *port* element of its own outer *block* element in a DXL multimodel.

- **initial attribute**: The *initial* attribute specifies the initial value of the *port* element.

### 3.2.2.4 The definition Element

```xml
<xsd:group name="DEFELEMENT">
  <xsd:choice>
    <xsd:element name="definition">
      <xsd:complexType>
        <xsd:attribute name="id" type="xsd:anyURI" use="required" />
        <xsd:attribute name="func" type="xsd:NCName" use="required" />
        <xsd:attribute name="lang" use="required">  
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:enumeration value="JavaScript" />
              <xsd:enumeration value="Python" />
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>
      </xsd:complexType>
    </xsd:element>
  </xsd:choice>
</xsd:group>
```

Figure 3-6. The Schema of Definition Element

The *definition* element defines the behavior for its own outer block element corresponding to the enclosing element. In a *definition* element, the actual program
codes for the simulation of the corresponding block element are located using their file name. Each code is composed of functions. A called function is referenced by an attribute func.

- **id attribute**: The id attribute specifies the file name of a document that provides the script code for the current block element. Because it can have paths, the id attribute specifies the location of a document that provides the script code for the current block element.

- **func attribute**: The func attribute specifies the starting function name to execute a file in the id attribute. Generally, because inputs of the function are the values of input port elements of the current block, its arguments are not required.

- **lang attribute**: The lang attribute specifies the language of a particular script in the DXL document.

### 3.2.2.5 The connect Element

```
<xsd:group name="DXLconnects">
  <xsd:choice>
    <xsd:element name="connect" minOccurs="0">
      <xsd:complexType>
        <xsd:group ref="ports" />
        <xsd:attribute name="id" type="xsd:IDREF" use="required" />
      </xsd:complexType>
    </xsd:element>
  </xsd:choice>
</xsd:group>
```

Figure 3-7. The Schema of Connect Element

An element connect connects a source port element and a target port element that are declared using an attribute source and target. It plays a role of a similar channel to a communication path. Generally, an output port element of the source block element becomes a source port element of connect element. And an input port element of a target block element becomes a target port element of a connect element.
• **id attribute:** The id attribute specifies the name of the connect element. The value of the id attribute generated by MXL-to-DXL translator has the meaning of a hierarchical structure because the name of the blocks in the higher layer in a multimodel are included.

### 3.2.2.6 The simulation Element

```xml
<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" />
    </xsd:complexType>
</xsd:element>
```

Figure 3-8. The Schema of Simulation Element

The simulation element includes the properties to execute generated simulation codes and customize the behavior of an actual simulation model relatively according to a physical time. So, the attributes of this element are different according to the simulation environment. This element is defined in the outermost DXL element.

• **start_time attribute:** The start_time attribute specifies the virtual starting time in simulation for a current DXL model.

• **end_time attribute:** The end_time attribute specifies the virtual ending time in simulation for a current DXL model.

• **delta_time attribute:** The delta_time attribute specifies the virtual unit time in simulation for a current DXL model. So, end_time-start_time/delta_time is the number of generated ticks for current simulation.

• **cycle_time attribute:** The cycle_time attribute is used to compensate the difference between virtual time and physical time. So, we can control the speed of moved or animated objects according to physical time.
3.3 The Semantic of DXL

3.3.1 The DXL Notation

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP_i</td>
<td>a set of input ports for a block i</td>
</tr>
<tr>
<td>IP_i(n)</td>
<td>the nth input port of a block i</td>
</tr>
<tr>
<td>OP_i</td>
<td>a set of output ports for a block i</td>
</tr>
<tr>
<td>OP_i(n)</td>
<td>the nth output port of a block i</td>
</tr>
<tr>
<td>OB_i(j)</td>
<td>a set of output blocks for the jth port of a block i</td>
</tr>
<tr>
<td>SEND(t, DATA(i), IP_k(j))</td>
<td>Current block sends data of ith output port to jth input port of block k after time t</td>
</tr>
</tbody>
</table>

Figure 3-9. Block Notation of DXL

Figure 3-9 defines a notation to describe the functionalities of DXL blocks. We can describe algorithmic semantic of blocks using a set theory and some pseudo codes. IP and OP mean input port and output port of a block. The order of the port is an argument and the identifier of its outer block is subscripted. So, the 1st and the 2nd input ports of block B1 are IP_{B1}(1) and IP_{B1}(2). The 1st and the 2nd output ports of block B1 are OP_{B1}(1) and OP_{B1}(2). OB means output blocks of current block. The input ports of these output blocks get the output after the current block finishes its computation. The order of an output port of the current block is an argument and the identifier of the current block is subscripted. So, the output blocks of the 1st and the 2nd output ports of the current block B1 are OB_{B1}(1) and OB_{B1}(2).

3.3.2 The Information Stream Mechanism of DXL

Figure 3-11 shows DXL contents for the left DXL model of Figure 3-10. Generally, a DXL model is composed of “block” elements and “connect” elements. A DXL block element is composed of “port” elements and “definition” elements but it
might include another DXL model when it represents a homogeneous multimodel. DXL block elements are connected by each connect element through input and output port elements. Input and output port elements can be included in a block element and each port element is connected to the corresponding output and input port element of other block elements for its data flow, which each connect element controls.

Figure 3-10. Diagrammatic Presentation of DXL

The pseudo code SEND is the code to support information streaming which operates according to the different modes. For example, the pseudo code SEND means “generate next event” based on discrete event scheduling methods [19, 27] in sequential simulation and “send message” based on message-passing protocols in distributed simulation. This information streaming abstraction approach used to generate different target codes on heterogeneous environments makes it easy to create DXL-to-simulation translator and integrate model components.

In Figure 3-9, the meaning of SEND pseudo code is “transfer DATA in an output port OP of a block B1 to an input port IP of a block B2 and generate next event B2, after time t passes” based on discrete event scheduling methods. If this model is based on message-passing protocols in distributed simulation, this SEND pseudo code means
“create \textbf{DATA} in an output port \textbf{OP} of a block \textbf{B1} as a message and send the message to an input port \textbf{IP} of a block \textbf{B2}, after time $t$ passes”

```xml
<DXL>
  <block id="B1" type="SYNC">
    <port id="B1.OP1" type="OUTPUT" target="B2.IP1">
    </port>
    <definition id="B1.js" func="B1_func" lang="JavaScript">
    </definition>
  </block>
  
  <block id="B2" type="SYNC">
    <port id="B2.IP1" type="INPUT" source="B1.OP1">
    </port>
    <definition id="B2.js" func="B2_func" lang="JavaScript">
    </definition>
  </block>

  <connect id="TR1">
    <port id="TR1.IP1" type="INPUT" source="B1.OP1">
    </port>
    <port id="TR1.OP1" type="OUTPUT" target="B2.IP1">
    </port>
  </connect>

  <simulation start_time="" end_time="" delta_time="" cycle_time="">
  </simulation>
</DXL>
```

Figure 3-11 Model of DXL for Figure 4-2 Example

In Figure 3-10, the meanings based on discrete event scheduling methods of SEND pseudo codes are as the follows:

- $\text{SEND}(t_1, \text{DATA}(1), \text{IP}_1(1))$: Transfer \text{DATA} in an output port \text{OP} of a block \text{B}_1 to an input port \text{IP}_1 of a block \text{B}_1 and \text{generate next event} \text{B}_1, after time $t_1$ passes.

- $\text{SEND}(t_m, \text{DATA}(1), \text{IP}_1(m))$: Transfer \text{DATA} in an output port \text{OP} of a block \text{B}_M to an input port \text{IP}_1 of a block \text{B}_1 and \text{generate next event} \text{B}_1, after time $t_m$ passes.

- $\text{SEND}(t_1, \text{DATA}(1), \text{IP}_1(1))$: Transfer \text{DATA} in an output port \text{OP} of a block \text{B}_1 to an input port \text{IP} of a block \text{B}_1 and \text{generate next event} \text{B}_1, after time $t_1$ passes.
- \textbf{SEND}(t_1, \text{DATA}(1), \text{IP}_n(1)) : Transfer \text{DATA} in an output port \text{OP}_n of a block \text{B}_1 to an input port \text{IP} of a block \text{B}_n and generate next event \text{B}_n, after time \text{t}_1 passes.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{example.png}
\caption{Examples of \textbf{SEND} Pseudo Code}
\end{figure}

In Figure 3-12, the meanings based on message-passing protocols in distributed simulation of \text{SEND} pseudo codes are as the follows:

- \textbf{SEND}(t_1, \text{DATA}(1), \text{IP}_1(1)) : Create \text{DATA} in an output port \text{OP}_1 of a block \text{B}_1 as a message and send the message to an input port \text{IP}_1 of a block \text{B}_1, after time \text{t}_1 passes.

- \textbf{SEND}(t_m, \text{DATA}(1), \text{IP}_m(m)) : Create \text{DATA} in an output port \text{OP}_m of a block \text{B}_m as a message and send the message to an input port \text{IP}_m of a block \text{B}_1, after time \text{t}_m passes.

- \textbf{SEND}(t_1, \text{DATA}(1), \text{IP}_1(1)) : Create \text{DATA} in an output port \text{OP}_1 of a block \text{B}_1 as a message and send the message to an input port \text{IP}_1 of a block \text{B}_1, after time \text{t}_1 passes.

- \textbf{SEND}(t_1, \text{DATA}(1), \text{IP}_n(1)) : Create \text{DATA} in an output port \text{OP}_n of a block \text{B}_1 as a message and send the message to an input port \text{IP}_1 of a block \text{B}_n, after time \text{t}_1 passes.

\subsection*{3.3.3 The XML-based Dataflow Semantic in DXL}

XML schema provides a syntax structure of an XML document. It can be used to check the validity of XML data when models exchange their data in a modeling and
simulation environment. Figure 3-13 describes two sorts of data-relationships in XML-based modeling and execution.

Figure 3-13. The XML-based Dataflow in DXL

Assume Java and JavaScript simulation programs to describe XML data exchange between different program languages on Web-based modeling and simulation environment. If we can support the data conversion between programming languages and XML data, it is possible to exchange the simulation data between different simulation programs. Basic relationships between programming and modeling data are described as follows:

- Actual programming Java data for simulation are converted to XML data based on their own XML schema.
- An output port sends its own XML data to the input port of a receiving block, where the XML data include their own XML schema link.
- After checking XML validity, the input port of the receiving block converts the XML data to actual JavaScript data.
The relationships between different models are the same as the basic relationships except for XML data conversion using XSLT. For data compatibility between two different models, we had to convert the XML data of model A to the XML data of model B. Because both models have their data with an XML format, the conversion has the same meaning as an XML document transformation if we use XSLT for our converter.

This usage of XML data with parameters makes it easy to use different programming languages for a model. Current DXL-to-simulation translator supports JavaScript and Java language using DOM. We will expand the supporting languages.

### 3.3.4 Synchronous and Asynchronous Input Semantic in DXL

**Algorithm for a block having synchronous inputs**

\[
\text{if}(\forall x \in \text{IP}_i, \ x \text{ has input data}) \text{ then}
\]

The computation of the function in a DXL block

\[
\text{For } \forall y \in \text{OP}_i, \text{ if}(y \text{ has output data}) \text{ then for } z \in \forall \text{OB}_i(y), \ \text{SEND}(t, \ \text{DATA}(y), z) \ \text{elseif}
\]

**Algorithm for block B3**

\[
\text{if } (\forall x \in \text{IP}_{B3}, \ x \text{ has input data}) \ 
\text{then}
\]

\[
\text{OP}_{B3}(1) \leftarrow \text{IP}_{B3}(1)/\text{IP}_{B3}(2)
\]

\[
\text{OP}_{B3}(2) \leftarrow \text{IP}_{B3}(1) \%\text{IP}_{B3}(2)
\]

\[
\text{SEND}(t_{B4}, \ \text{DATA}(\text{OP}_{B3}(1)), B4)
\]

\[
\text{SEND}(t_{B5}, \ \text{DATA}(\text{OP}_{B3}(2)), B5)
\]

**Figure 3-14. Example of Synchronous Inputs**

If a DXL block has more than one input, there are two options, **synchronous** and **asynchronous**, in DXL. These two properties can not only support the properties of
multiple other model types but also expand the flexibility and efficiency of MXL-to-DXL translation.

```
<DXL>
<block id="B1" type="SYNC">
  <port id="B1.OP1" type="OUTPUT" target="B3.IP1"></port>
  <definition id="B1.js" func="B1_func" lang="JavaScript"></definition>
</block>

<block id="B2" type="SYNC">
  <port id="B2.OP1" type="OUTPUT" target="B3.IP2"></port>
  <definition id="B2.js" func="B2_func" lang="JavaScript"></definition>
</block>

<block id="B3" type="SYNC">
  <port id="B3.IP1" type="INPUT" source="B1.OP1"></port>
  <port id="B3.IP2" type="INPUT" source="B2.OP1"></port>
  <port id="B3.OP1" type="OUTPUT" target="B4.OP1"></port>
  <port id="B3.OP2" type="OUTPUT" target="B5.OP1"></port>
  <definition id="B3.js" func="divide_func" lang="JavaScript"></definition>
</block>

<block id="B4" type="SYNC">
  <port id="B4.IP1" type="INPUT" source="B3.OP1"></port>
  <definition id="B4.js" func="B4_func" lang="JavaScript"></definition>
</block>

<block id="B5" type="SYNC">
  <port id="B5.IP1" type="INPUT" source="B3.OP2"></port>
  <definition id="B5.js" func="B5_func" lang="JavaScript"></definition>
</block>

<connect id="TR1">
  <port id="TR1.IP1" type="INPUT" source="B1.OP1"></port>
  <port id="TR1.OP1" type="OUTPUT" target="B3.IP1"></port>
</connect>

<connect id="TR2">
  <port id="TR2.IP1" type="INPUT" source="B2.OP1"></port>
  <port id="TR2.OP1" type="OUTPUT" target="B3.IP2"></port>
</connect>

<connect id="TR3">
  <port id="TR3.IP1" type="INPUT" source="B3.OP1"></port>
  <port id="TR3.OP1" type="OUTPUT" target="B4.OP1"></port>
</connect>

...<simulation start_time="" end_time="" delta_time="" cycle_time""></simulation>
</DXL>
```

Figure 3-15. Source of DXL for Figure 3-14

If the DXL block has the same property as a programming function, this means that
the DXL block should execute its code after it has all its inputs like programming
functions to do. In DXL, the block is called “The block has synchronous input property”.

In short, the block is a **synchronous block** in DXL.

Otherwise, the DXL block has the same property as a network node, this means that the DXL block executes its code only if it has one of its inputs like a network node service. In DXL, we say that “The block has asynchronous input property”. In short, the block is an asynchronous block in DXL.

---

**Algorithm for a block having asynchronous inputs**

if (for \( \exists x \in IP_i \), \( x \) has input data) then

The computation of the function in a DXL block

For \( \forall y \in OP_i \), if \( y \) has output data then for \( z \in \forall OB_i(y) \), SEND(t, DATA(y), z) elseif

elseif

---

**Algorithm for block B3**

if (IP_{B3}(1) has input data) then

\( OP_{B3}(2) \leftarrow IP_{B3}(1) \)

SEND(t_{B5}, DATA(OP_{B3}(2)), B5)

elseif

if (IP_{B3}(2) has input data) then

\( OP_{B3}(1) \leftarrow IP_{B3}(2) \)

SEND(t_{B4}, DATA(OP_{B3}(1)), B4)

elseif

---

Figure 3-16. Example of Asynchronous Inputs

### 3.3.4.1 Synchronous Input Property in DXL

Block B3 in Figure 3-14 divides the output of block B1 with the output of block B2 and then sends its quotient to block B4 and remainder to block B5. To make the division, block B3 should have two inputs at the same time, which means that it is a synchronous block in DXL. Note that the block has two output ports to support a different output at the
same time unlike the functions of traditional programming languages, which have just one output. The right part of Figure 3-14 describes the processing algorithm of block B3.

```xml
<DXL>
  <block id="B1" type="SYNC">
    <port id="B1.OP1" type="OUTPUT" target="B3.IP1"></port>
    <definition id="B1.js" func="B1_func" lang="JavaScript"></definition>
  </block>
  <block id="B2" type="SYNC">
    <port id="B2.OP1" type="OUTPUT" target="B3.IP2"></port>
    <definition id="B2.js" func="B2_func" lang="JavaScript"></definition>
  </block>
  <block id="B3" type="ASYNC">
    <port id="B3.IP1" type="INPUT" source="B1.OP1"></port>
    <port id="B3.IP2" type="INPUT" source="B2.OP1"></port>
    <port id="B3.OP1" type="OUTPUT" target="B4.OP1"></port>
    <port id="B3.OP2" type="OUTPUT" target="B5.OP1"></port>
    <definition id="B3.js" func="switch_func" lang="JavaScript"></definition>
  </block>
  <block id="B4" type="SYNC">
    <port id="B4.IP1" type="INPUT" source="B3.OP1"></port>
    <definition id="B4.js" func="B4_func" lang="JavaScript"></definition>
  </block>
  <block id="B5" type="SYNC">
    <port id="B5.IP1" type="INPUT" source="B3.OP2"></port>
    <definition id="B5.js" func="B5_func" lang="JavaScript"></definition>
  </block>
  <connect id="TR1">
    <port id="TR1.IP1" type="INPUT" source="B1.OP1"></port>
    <port id="TR1.OP1" type="OUTPUT" target="B3.IP1"></port>
  </connect>
  <connect id="TR2">
    <port id="TR2.IP1" type="INPUT" source="B2.OP1"></port>
    <port id="TR2.OP1" type="OUTPUT" target="B3.IP2"></port>
  </connect>
  ...
  <connect id="TR4">
    <port id="TR4.IP1" type="INPUT" source="B3.OP2"></port>
    <port id="TR4.OP1" type="OUTPUT" target="B5.IP1"></port>
  </connect>
  <simulation start_time="" end_time="" delta_time="" cycle_time=""></simulation>
</DXL>
```

Figure 3-17. Source of DXL for Figure 3-16
3.3.4.2 Asynchronous Input Property in DXL

In Figure 3-16, block B3 plays a role in switching input data. It is not necessary to receive both inputs to relay its data. In other words, the block just relays its input to its next block whenever it receives the input. The block should therefore have an asynchronous input property. The switch block algorithm is described in the right part of Figure 3-16. In DXL, the above combinations of simple blocks and their properties create an actual model for a system.
CHAPTER 4
MULTIMODELING

4.1 Multimodeling through MXL and DXL

Multimodeling is a modeling process to model a system at multiple levels of abstractions and a multimodel can be created to represent and analyze a complex system through this process. Multimodeling is achieved through functional composition and ports coupling. How can we achieve the functional composition and ports coupling? How can we achieve multimodeling based on the two methods? In the RUBE framework, the functional composition and ports coupling are described through MXL for heterogeneous multimodeling and DXL for homogeneous multimodeling.

Figure 4-1. Multimodel Structure of MXL

MXL [10] is an XML-based dynamic modeling language to represent traditional model types on our RUBE framework. For MXL, the functional elements for each model type are clearly identified and then entry points of the functional elements are defined as ports. The functional elements are model components that behave as functions, or where functions naturally fit as part of the model description. Figure 4-1 shows MXL multimodel structure of MXL. Multimodeling is permitted wherever the appropriate model component can be “extended” or “expanded” to another model whose outermost
definition is a function. These couplings achieve multimodeling through inserting one function inside of another (inter-level coupling) or connecting the function to another (intra-level coupling).

Figure 4-2. Heterogeneous and Homogeneous Models

The left part of Figure 4-2 shows an abstract view for our MXL. Each model can be a heterogeneous model and is coupled using ports, which are interfaces for compatibility of input and output types. Each model in the abstract MXL can be defined as a traditional heterogeneous model by XML, which are FSM, FBM, Petri net, etc. Then each MXL model can be translated into the contemporary DXL models by our model-specific translators using XSLT. As the last process, our DXL-to-simulation translator generates program language-specific simulation programs. The right part of Figure 4-2 shows a simple view for the translated DXL for the left part of Figure 4-2. Each translated DXL model is connected through input and output ports.

4.2 MXL-to-DXL Translation

4.2.1 Basic MXL-to-DXL Translation Example

MXL models are heterogeneous models. Each heterogeneous model is defined by the corresponding MXL. The MXL model of Figure 4-3 contains \texttt{fbm} element and \texttt{simulation} element as sub-elements. \texttt{fbm} element represents a Functional Block
Model (FBM). It has two elements, namely block element and trace element. Each block element has an attribute, id, to specify the names of blocks. In addition, the element has the information of port element, input and/or output, as well as its functionality, script element.

```xml
<MXL>
  <fbm id="MXL">
    <block id="F1">
      <output id="F1_outports_integer1" datatype="Integer" index="0"/>
      <script lang="JavaScript" src="input.js" func="gen"/>
    </block>
    <block id="F2">
      <input id="F2_inports_integer1" datatype="Integer" index="0"/>
      <output id="F2_outports_string1" datatype="String" index="0"/>
      <fsm id="FSM_F2" src="FSM_F2.xml"/>
    </block>
    <block id="F3">
      <input id="F3_inports_string1" datatype="String" index="0"/>
      <script lang="JavaScript" src="input.js" func="disp"/>
    </block>
    <trace from="F1_outports_integer1" to="F2_inports_integer1"/>
    <trace from="F2_outports_string1" to="F3_inports_string1"/>
  </fbm>
  <simulation start_time="0" end_time="10" delta_time="0.1" cycle_time="0.1" />
</MXL>
```

Figure 4-3. Model of MXL as the Wrapper of the Four-Stroke Gasoline Engine

Each input and output element contain id, datatype, and index, which define the name of input/output, data type of input/output, and the order of input/output, respectively. And script element has lang, src, and func, the sort of program language, the name of script file and its location, and start function to be executed, respectively. If a block element contains another model type, which is called a multimodel, the model type and its MXL can be specified in the block element. trace element has from and to attributes denoting connectivity between blocks. The value of the elements is the id attribute of port elements. simulation element has specific information for executing a certain dynamic model such as start_time, end_time,
and \( \text{delta\_time} \). These elements are used for compensation of the difference between virtual time and physical time.

```xml
<MXL>
  <fsm id="FSM_F2">
    <input id="F2_inports_integer1" datatype="Integer" index="0"/>
    <output id="F2_outports_string1" datatype="String" index="0"/>

    <state id="off" start="true">
      <script lang="JavaScript" src="input.js" func="off"/>
    </state>
    <state id="compression">
      <script lang="JavaScript" src="input.js" func="compression"/>
    </state>
    <state id="ignition">
      <script lang="JavaScript" src="input.js" func="ignition"/>
    </state>
    <state id="expansion">
      <script lang="JavaScript" src="input.js" func="expansion"/>
    </state>
    <state id="exhaustion">
      <script lang="JavaScript" src="input.js" func="exhaustion"/>
    </state>

    <transition from="off" to="compression">
      <script lang="JavaScript" src="input.js" func="off2Compression"/>
    </transition>
    <transition from="compression" to="ignition">
      <script lang="JavaScript" src="input.js" func="compression2Ignition"/>
    </transition>
    <transition from="ignition" to="off">
      <script lang="JavaScript" src="input.js" func="ignition2Off"/>
    </transition>
    <transition from="ignition" to="expansion">
      <script lang="JavaScript" src="input.js" func="ignition2Expansion"/>
    </transition>
    <transition from="expansion" to="exhaustion">
      <script lang="JavaScript" src="input.js" func="expansion2Exhaustion"/>
    </transition>
    <transition from="exhaustion" to="compression">
      <script lang="JavaScript" src="input.js" func="exhaustion2Compression"/>
    </transition>
  </fsm>
</MXL>
```

Figure 4-4. Model of MXL for the Four-Stroke Gasoline Engine

Figure 4-4 depicts internal dynamic model type, Finite State Machine (FSM), which block F2 contains. In this case, the MXL represents the FSM of the Four-Stroke Gasoline Engine in Figure 4-7. The MXL model for FSM has state and transition
elements as sub-elements. Each state and transition has its id element for the name of the elements and script information for the execution of simulation. It represents topological connectivity and state transition through state and transition elements. From and to attributes of transition elements are similar to trace elements of FBM.

```xml
<DXL>
  <block id="F1" type="SYNC">
    <port id="F1.F1_outports_integer1" type="OUTPUT"
      target="F2.F2_inports_integer1" data_type="Integer"></port>
    <definition id="input.js" func="gen" lang="JavaScript"></definition>
  </block>
  <block id="F2" type="SYNC">
    <port id="F2.F2_inports_integer1" type="INPUT"
      source="F1.F1_outports_integer1" target="FSM_F2_INPUT.IP1" data_type="Integer"></port>
    <port id="F2.F2_outports_string1" type="OUTPUT"
      target="F3.F3_inports_string1" data_type="String"></port>
    <DXL id="FSM_F2">
      <block id="FSM_F2_INPUT" type="SYNC" internal="YES">
        <port id="FSM_F2_INPUT.IP1" type="INPUT" source="F2.F2_inports_integer1" data_type="Integer"></port>
        <port id="FSM_F2_INPUT.IP2" type="INPUT" source="FSM_F2_OUT.OP1" data_type="String"></port>
      </block>
      <block id="FSM_F2_OUT" type="ASYNC" internal="YES">
        <port id="FSM_F2_OUT.OP1" type="OUTPUT" target="FSM_F2_INPUT.IP2" data_type="String"></port>
      </block>
    </DXL>
  </block>
  <block id="F3" type="SYNC">
    <port id="F3.F3_inports_string1" type="INPUT" source="F2.F2_outports_string1" data_type="String"></port>
    <definition id="input.js" func="disp" lang="JavaScript"></definition>
  </block>
  <connect id="F1_outports_integer1ToF2_inports_integer1">
    <port id="F1_outports_integer1ToF2_inports_integer1.OP1" type="INPUT" source="F1.F1_outports_integer1"></port>
    <port id="F1_outports_integer1ToF2_inports_integer1.IP1" type="OUTPUT" source="F2.F2_inports_integer1"></port>
  </connect>
  ...
</DXL>
```

Figure 4-5. Model of DXL for the Four-Stroke Gasoline Engine

There are similar characteristics between FBM and FSM in terms of topology. But, the fact that FBM represent dataflow while FSM represent control flow make it difficult
to integrate two models through composition or decomposition. Each model is transformed into an independent DXL model through MXL-to-DXL translator.

Using MXL-to-DXL Translator, the MXL file can be translated into DXL, which is a homogeneous block diagram modeling language consisting of connect, block, and port elements. Figure 4-5 shows the generated DXL model from the MXL as shown in Figures 4-3 and 4-4. When an FSM model is inserted into a block of an FBM model, the FSM model is used for representation of the behavior or dynamics of the block in the FBM model. Ultimately a JavaScript-based simulation code for the example to represent and execute the multimodel is generated through DXL2JavaScript translator.

4.2.2 FBM-to-DXL Translation

Functional Block Model (FBM) is composed of functional elements where functions, along with inputs and outputs, are often depicted in a “block” form. An arbitrary number of blocks can be coupled together to form an FBM. This FBM is similar to DXL except for synchronous and asynchronous inputs of DXL. Therefore, functions of FBM are translated into DXL blocks and its inputs and outputs are translated into DXL input and output ports. Because each block of traditional FBM has a pure function, all inputs of the function should be valid before the block is executed. This means that all FBM blocks have synchronous property for inputs.

Figure 4-6. Translation an FBM into a DXL Model
An FBM for the differential equation \( x'' = -ax' \) is described in the left part of Figure 4-6. This FBM is composed of four blocks: two integrators, one multiplier, and one constant. Because each block of traditional FBM has a synchronous input property, these blocks are translated into blocks having synchronous inputs in DXL. Its arrows are translated into connectors of DXL. This FBM also has a continuous simulation property. To support the property, a start block of the DXL model is generated every simulation unit time.

### 4.2.3 FSM-to-DXL Translation

![Diagram](image)

Figure 4-7. Translation an FSM into a DXL Model

Finite State Machine (FSM) has states and transitions. A state represents the current condition or “snapshot” of a system for some length of time. Transitions enable the system to move from one state to another during the simulation while under the control of the system input. The basic rule of translating FSM into DXL is that all functional elements are translated into DXL blocks. The transitions of FSM are predicates under the system input and can, therefore, be translated into DXL blocks which have the same type of input ports as the system input and boolean type of output ports deciding whether the predicates are true or false. Since states have the functional properties that they can...
access the system input, all states are translated into DXL blocks. In addition, we need the special block to control the system input.

Figure 4-7 shows a FSM modeling a four-stroke gasoline engine with four phases such as compression, ignition, expansion, and exhaustion. The key point that makes FSM different from FBM is a state-based model. To translate a state-based model into a function-based model, all states and transitions are translated into DXL blocks. Then DXL connectors and block properties control the semantic of the state-based model. The right part of Figure 4-7 shows a translated DXL model for the four-stroke gasoline engine. INPUT and OUTPUT blocks are generated to control the FSM semantic by our translator. The right blocks next to INPUT blocks are translated transition blocks and the right blocks next to OUTPUT blocks are translated state blocks.

4.2.4 Petri Net-to-DXL Translation

Petri nets are used primarily for studying the dynamic concurrent behavior of network-based systems where there is a discrete flow. The components of Petri nets are a finite set of places and transition. The places of Petri nets are conditions for firing their transitions and have tokens. The functional elements of Petri nets are transitions and places. Therefore, both transitions and places are translated into DXL blocks that have the same type of input and output ports as tokens. Especially, the transitions of Petri nets can fire only after all input places have tokens, which means all translated transition blocks have synchronous input property.
An assembly operation in a flexible manufacturing system is modeled by the left Petri net in Figure 4-8. M1 and M2 are machines for milling, cutting, or another similar operation. A1 represents assembling parts that come from M1 and M2. The tokens in the Petri net represent the materials that flow through those machine lines. The translations in the Petri net can fire after tokens arrive from all their input places. Those transitions are therefore translated into synchronous DXL blocks, as shown in the right part of Figure 10. The places in the Petri net are also translated into DXL blocks, but since those places can receive the tokens from different transitions at different time, they should be asynchronous DXL blocks. The blocks having more than two tokens generate their next blocks and themselves with their next events.

### 4.3 Multimodeling in DXL

For DXL, multimodeling is defined by defining a block circuit inside of a block, and continuing recursively as needed. Ports are coupled together by ensuring matching data types for each connecting port on each connector. General DXL methodology in transforming heterogeneous models (MXL models) into homogeneous models (DXL models) is as the following:

- Transform sub-models in a MXL multimodel into DXL models.
- Incorporate each of the transformed models according to port coupling.
In this thesis, the flexibility and efficiency for multimodeling of DXL, based on four types of refinements shown in chapter 2, are described.

4.3.1 Multimodeling among Homogeneous Models in DXL

In Figure 4-9, the upper model is a two-level functional hierarchy where function \( f \) is defined in terms of a composition of three other functions \( f_1, f_2, \) and \( f_3 \). The lower model shows multimodeling in DXL for upper model. Because DXL is specifically based on a functional block diagram, FBM models in Figure 4-9 can easily be refined only if the number and types of ports are matched.

\[
O = f(i_1, i_2) = f_3(f_1(i_1), f_2(i_2))
\]

![Figure 4-9. Multimodeling between Functional Block Models in DXL](image)

In Figure 4-10, the upper model is a two-level finite state hierarchy where state \( s_1 \) is defined in terms of another FSA. The lower model shows the transformed DXL model for the two-level finite state hierarchy. If heterogeneous models (MXL models) are FSA, there is a notice for supporting multimodeling. All FSAs should have this kind of
functional box around them to represent explicit external input and output semantics even though the functional box is not always explicit in the FSA graphical representation.

Figure 4-10. Multimodeling between FSA Models in DXL

While FSA represent behavior or dynamics of another component in multimodeling, a FSA can be inserted in it to represent a state more specifically. In that case,
multimodeling between FSAs should be supported, but there is a difficulty of the integration of two FSAs because of the implicit expression of input and output in FSA.

But, in DXL, states and transitions in FSA are transformed into the same blocks. And, specific blocks, INPUT and OUTPUT blocks, are generated and they control state and transition blocks. The programming codes of the INPUT and OUTPUT blocks are automatically generated by MXL-to-DXL translator. In Figure 4-10, after a sub-model, the lower FSA model, is transformed into a DXL model, it is inserted in its upper state block and then connected with external input and output ports using connect elements in DXL.
4.3.2 Multimodeling among Heterogeneous Models in DXL

In Figure 4-11, the upper model is the same model as Figure 2-4. The sub-model in this figure is an FSA that defines the internal state transitions associated with function $f$. This approach is the same one as that for a four stroke gasoline engine model in Figure 4-7. The generated DXL model for an FSA is inserted in function $f$. Because the number and types of input and output of the function $f$ are the same as the input and output ports of the DXL model, the integration can be accomplished easily through port matching.

![Diagram of multimodeling among heterogeneous models in DXL](image)

Figure 4-12. Multimodeling for an FSM Including an FBM in DXL

In Figure 4-12, a state of an FSM decomposes into a FBM to represent internal functionality. The state in an FSM is different from a FBM from the viewpoint of a system. While, in a FSM, input and output are not specified explicitly, control flow is
unclear in an FBM. But, in a DXL model, the input and output of transformed FSM states are specified explicitly. That makes it easier for the integration of the components of a FSM and the components of a FBM. The lower model in Figure 4-12 is a transformed DXL multimodel. A sub-layer of the model, a transformed DXL model for an FBM, is connected to the input and output ports of the outer state block $s_1$. 
5.1 Concept of Simulation Code

5.1.1 The Structure of a DXL Block Class

The final simulation code generated by DXL-to-SIM translator is executed using the event scheduling method supported by SimPack.

Each DXL block is defined as class in the simulation code. The class is composed of two methods such as initialization and controller methods. The initialization method creates the internal variables and initializes the port data and the function of its own DXL block. The controller processes the output data in its ports and calls its block function inserted by a user in MXML.

```
class fsm_1_SUB_S1_class:
    class __init__(self):
        self.input = []
        self.output = []
        for i in xrange(self.NUM_INPUT_PORTS):
            self.input.append(None)
        for i in xrange(self.NUM_OUTPUT_PORTS):
            self.output.append(None)
        self.function_Controller = self.S3_Controller
        self.user_function = S3

    def S1_Controller(self):
        self.user_function(self.input,
                          self.output)
        if(self.output[0] != None):
            self.out_ports_check[0] = TRUE
```

Figure 5-1. Structure of a DXL class in Python
In JavaScript, the structure of a DXL block class is the same as Python version except for the controller method is defined out the class. Figure 5-1 and Figure 5-2 show the structure of a DXL block class and source code for Python and JavaScript.

**5.1.2 The Structure of a DXL Block Class**

DXL block objects are scheduled as events on future event list of SimPack for JavaScript or Python. The simulation code uses schedule function to schedule the events and next_event function to extract next event.

An event scheduling is defined as executing code that contains an event loop where events are posted and then checked according to minimum time [21, 46, 70, 72]. An output produced from DXL using JavaDOM is structurally a simulation code using an event-based scheduling method.
The simulation program has a single event loop for scheduling DXL blocks on a future event list according to their event times. After executing a block, target blocks outgoing from that block are yielded as new events.

Figure 5-3. Event-based Simulation

If the block has some other model, its start block can also be yielded as a new event. Those events are scheduled on a future event list. An event scheduling, on which simulation code is based, is described in Figure 5-3.

An event having the smallest time on the future event list is executed. Each event is a block having a Python code in simple model or some other model in multimodel in a definition element of a DXL model. Calling next event method provided by a SimPackJ/S toolkit can get the events having the earliest time.

According to each event, the corresponding function, which is a block of a DXL model, is called. After the function is called, next blocks become new events. Those
blocks are the output blocks and a start block in an included model of a current executing block.

A translator using DOM can see the output blocks by finding blocks having input ports connected to output ports of the block. The output blocks or a new start block can be scheduled by calling the schedule method provided by a SimPackJ/S toolkit.

Figure 5-4. Structure of Simulation File

The structure of JavaScript file for simulation is shown in Figure 5-4. This executable simulation code is conceptually composed of three parts such as user input code, SimpackJ/S code, and control code created by DXL translator. On the other hand, Figure 5-4 shows that this code structurally consists of part 1, part 2, and part 3. Part 1 is an initialization part such as variable declaration and initialization. Part 2 is a loop run until simulation end time. Part 3 is a user input JavaScript code. In the loop of part 2,
calling function for a current event and scheduling function for output and inner block with next events are included.

5.2 MXL, DXL, AND SIMULATION Translation

5.2.1 A MXL Multimodel for a Light Bulb Example

A light bulb example using a FBM and a FSM is described as a through application example for the MXL-DXL-Simulation. The FBM represents the generation of input and the display of output for the light bulb example. The FSM represents the dynamics of the light bulb example.

![Figure 5-5. Model of FBM for a Light Bulb Example](image)

![Figure 5-6. Model of FBM for a Light Bulb Example](image)

The 2D diagram of the FBM with three functions and two traces is depicted in Figure 5-5. The F1 is the block that generates the input for an FSM. The F2 includes the FSM for the dynamics of the example. The output of F2 is displayed in the F3. The 2D diagram of the FSM representing the dynamics of the light bulb example is showed in
Figure 5-6. The S1 is a start state. The S2 and the S3 represent “ON” and “OFF” states of a light bulb respectively.

![Diagram of light bulb example](image)

**Figure 5-7. 2D Diagram for a Light Bulb Example**

```xml
<MXL>
  <fbm id = "fbm_ex">
    <block id="F1">
      <output id="FBM_F1_output" datatype="Integer" index="0"/>
      <script lang="Python" src="F1.js" func="f1_func"/>
    </block>
    <block id="F2">
      <input id="FBM_F2_input" datatype="Integer" index="0"/>
      <output id="FBM_F2_output" datatype="String" index="0"/>
      <fsm id="fsm_1" src = "fsm_1.mxl"/>
    </block>
    <block id="F3">
      <input id="FBM_F3_input" datatype="String" index="0"/>
      <script lang="Python" src="F3.js" func="f3_func"/>
    </block>
    <trace id="TR12" from = "FBM_F1_output" to = "FBM_F2_input"/>
    <trace id="TR23" from = "FBM_F2_output" to = "FBM_F2_input"/>
  </fbm>
  <simulation start_time = "0" end_time = "10" delta_time = "1" cycle_time = "0.1"/>
</MXL>
```

**Figure 5-8. Light Bulb Example for MXL of an FBM**

Figure 5-7 depicts an abstract level 2D diagram for our light bulb example that is composed of FBM with three functions and FSM with three states inside the second function of the FBM. The multimodel represented in MXL is shown in Figure 5-8 and
Figure 5-9, where an FSM is contained inside the node element whose id attribute value is “F2”.

```xml
<MXL>
  <input id="FSM_input" datatype="Integer" index="0"/>
  <output id="FSM_output" datatype="String" index="0"/>
  <fsm id="fsm_ex">
    <state id="S1" start="true">
      <script lang="Javascript" src="S1.js" func="S1"/>
    </state>
    <state id="S2">
      <script lang="Javascript" src="S2.js" func="S2"/>
    </state>
    <state id="S3">
      <script lang="Javascript" src="S3.js" func="S3"/>
    </state>
    <transition id="t1" from = "S1" to = "S1">
      <script lang="Javascript" src = "T1.js" func = "t1_func"/>
    </transition>
    <transition id="t3" from = "S2" to = "S2">
      <script lang="Javascript" src = "T2.js" func = "t2_func"/>
    </transition>
    <transition id="t5" from = "S3" to = "S3">
      <script lang="Javascript" src = "T3.js" func = "t3_func"/>
    </transition>
    <transition id="t2" from = "S1" to = "S2">
      <script lang="Javascript" src = "T4.js" func = "t3_func"/>
    </transition>
    <transition id="t4" from = "S2" to = "S3">
      <script lang="Javascript" src = "T6.js" func = "t6_func"/>
    </transition>
  </fsm>
</MXL>
```

Figure 5-9. Light Bulb Example for MXL of an FSM

5.2.2 A DXL Multimodel and Simulation for a Light Bulb Example

![Diagram of a light bulb example for DXL FBM model]

Figure 5-10. Light Bulb Example for DXL FBM Model
Figure 5-11. Source of FBM Model in DXL and JavaScript

An example of an FBM model in DXL, which has three blocks and two connects, is depicted in Figure 5-10. This DXL model for FBM will be used as data input generator, model container, and output reporter. Block f1 will be an input generator and block f3 play a role of an output reporter. A DXL model for FSM will be included in block f2, which functions as a model container.

Figure 5-11 is a DXL result from an MXL FBM by an MXL-to-DXL translator and content of an executable JavaScript file for simulation. Each block in FBM is translated into each block in DXL. So, this example consists of three blocks such as f1, f2, and f3 sequentially connected. The block f2 will include another DXL model, a light bulb FSM. In the left DXL part, an attribute id in a definition element has a name of a JavaScript file for the corresponding block.
A DXL example of an FSM model, which has three states and three transitions, is depicted in Figure 5-12. Each state and transition is translated into each block in DXL. States S1, S2, and S3 are translated into blocks s1, s2, and s3 in DXL. Transitions t1, t2, t3, t4, t5, and t6 are translated into blocks t1, t2, t3, t4, t5, and t6 in DXL.

The left part of Figure 5-13 is a DXL result from an MXL FSM by an MXL-to-DXL translator and the right part is an executable simulation code for a DXL FSM model. Only blocks are translated into DXL blocks in DXL-FBM. On the other hand, both states and transitions in FSM are translated into DXL blocks.
Figure 5-13. Source of FBM Model in DXL and JavaScript

So, blocks meaning transitions as well as states should have JavaScript code. An example of a multimodel that incorporates the DXL models for FSM and FBM as described in Figures 5-10 and 5-12 is depicted in Figure 5-14. Block f2 in Figure 5-14 contains the FSM model that is translated into the corresponding example. So, Block f1 generates block f2 with a next event. Then block f2 generates INPUT, which is an input deliverer according to current state in the FSM model, with a next event. After the inner DXL model is executed, the f2 generates f3.

Figure 5-15 shows the content of a DXL file for our light bulb example as well as parts of a simulation code for JavaScript translated from it. The left side of Figure 5-15 is a DXL code having a block f1, f2, and f3. A B1 block is a function f1 func that generates input data for the FSM model included in a block f2.

An f2 block is a function f2 func that generates a start block of the FSM model with a next event. An f3 block is a function f3 func that reports the current state in the FSM
model included in a block f2. Each block has a function name to be called in a func
attribute in a definition element or some other model.

Figure 5-14. Light Bulb Example for DXL Multimodel

The right side of Figure 5-15 is a simulation code for JavaScript produced from the
DXL file of the left side of Figure 5-15 through a translator using DOM. In the right side
of the figure, the function names are called in a “switch statement” in an “update
function,” which is called in a “while statement” of a main function.

Figure 5-16 is a partial output graph for our light bulb multimodel. This output graph
shows the change of current state in the FSM model included in block f2 for input data,
Figure 5-15. Source of Multimodel in DXL and JavaScript

```xml
<xml version="1.0" encoding="UTF-8"/>
<root>
  <block id="1" type="INPUT" source="s1">
    <port id="p1" type="OUTPUT" target="f1">
      <definition id="f1_input_func"/>
    </port>
  </block>
  <block id="2" type="INPUT" source="s2">
    <port id="p2" type="OUTPUT" target="f2">
      <definition id="f2_input_func"/>
    </port>
  </block>
  <block id="3" type="OUTPUT" target="f1">
    <port id="p3" type="INPUT" target="s1">
      <definition id="f1_output_func"/>
    </port>
    <port id="p4" type="INPUT" target="s2">
      <definition id="f2_output_func"/>
    </port>
  </block>
  <block id="4" type="OUTPUT" target="f2">
    <port id="p5" type="INPUT" target="s1">
      <definition id="f1_output_func"/>
    </port>
    <port id="p6" type="INPUT" target="s2">
      <definition id="f2_output_func"/>
    </port>
  </block>
  <root>
</xml>
```

function main() {
  while (sim_time() < end_time) {
    next_event(event, block_token);
    if (sim_time() == last_time) {
      last_time = sim_time();
    }
    update_block(event, block_token);
  }
}

function update_block(event, block_token) {
  var l1, l2, block_num = block_token.str[0];
  switch (event.value) {
    case 0: f1_func(block_token, layer);
      break;
    case 1: f2_func(block_token, layer);
      break;
    case 2: f3_func(block_token, layer);
      break;
    case 3: f4_func(block_token, layer);
      break;
    case 4: f5_func(block_token, layer);
      break;
    case 5: f6_func(block_token, layer);
      break;
    default: break;
  }
}

Figure 5-16. Partial Result of Simulation
CHAPTER 6
CONCLUSION

6.1 Summary of Results

DXL, an XML-based low-level functional specification language, was designed in this thesis. The role of DXL helps to support the independence between traditional heterogeneous model types and actual programming languages. This independence provides model designers to more easily create and integrate multiple models. DXL also plays a role of moving fast and easily from a sequential simulation environment to a Web-based simulation environment.

The contributions of this thesis in the field of modeling and simulation can be summarized as follows:

6.1.1 Web-based Modeling and Simulation by Using XML to Represent Simulation Models

MXL and DXL use XML to support model composability and reusability for Web-based modeling and simulation. Heterogeneous traditional models are defined using MXL and homogeneous primitive models are defined using DXL. Each XML-based model might include its own type of zero or more XML-based models as well as its own atomic XML elements. MXL might support a heterogeneous multimodeling and DXL might support a homogeneous multimodeling through the hierarchy of the models. Because both languages use XML, it is easy to transform a heterogeneous model to a homogeneous model through XSLT.
6.1.2 Multimodeling Using MXL, DXL, and an MXL-DXL bridge

The approach of this thesis supports multimodeling through an MXL-DXL bridge where heterogeneous multimodels are specified with MXL and then translated into the homogeneous multimodels using a low-level functional specification language, which is DXL. In this thesis, multimodeling is approached through transforming heterogeneous models into homogeneous models, and functional composition with coupled ports of the same data type.

6.1.3 Flexible Diagrammatic Presentation of Functionality for DXL Block

In DXL, models are specified by connecting inputs and outputs of primitive models such as multipliers or adders. Each block can be encoded in one of the following languages: JavaScript or Python. Multiple programming languages become the target codes for DXL-to-Simulation translators and an actual data flow for simulation is achieved if we use XML data and schema. DXL supports the combination of lower-level models of subsystems for complex systems hierarchically which means homogeneous multimodeling.

6.1.4 Ports-based Modeling and Simulation Capable of Encoding Documents rather than Data

Ports constitute the interface that defines the boundary of components or subsystems in a system configuration. MXL and DXL ports are used for port coupling, where all ports match in number and data type. MXL ports are for conceptual multimodeling in rube framework and DXL ports are there to maintain the conceptual multimodeling and support the multimodel execution. Because of the XML-based environment, DXL ports are capable of encoding documents rather than data, or more specifically, XML documents. Thus, a DXL or MXL model supports XML “information
streaming” and not only streaming of simple data types. The “types” are accompanied by XML schemata that define the typing structure.

6.1.5 Explicit Support for Discrete Event and Continuous Simulation

For high-fidelity simulation of multidisciplinary systems, hybrid modeling and simulation is required to support a combination of continuous time physical behavior and events occurring at discrete time and space. DXL explicitly supports simulation in terms of the development of commonly used simulation blocks found in both discrete-event and continuous simulation.

6.2 Future Research

6.2.1 Support of Different Model Types for Multimodeling

We support FBM and FSM for multimodeling now. We plan the design of MXL and MXL-to-DXL translators on other model types for our multimodeling. Those model types are the following types: EQN (difference and differential equations), QM (Queuing Model), PETRI (Petri nets), MARKOV (Markov Models), and SD (System Dynamics Models). Currently, Petri nets-to-DXL and QM-to-DXL translation schemes are accomplished for supporting multimodeling.

To support a variety of model types and multi-layered models, there is a problem for maintaining a data consistency among a variety of models because their use of different data. To maintain the consistency, we need to design and formalize a MXL-to-DXL translation scheme.

6.2.2 Distributed Simulation

The basic architecture used in Web-based modeling and simulation is described in the left part of Figure 6-1. DXL models are translated into specific program languages. But to support multiple programs created by different program languages, we need
method invocations, such as traditional distributed systems. Because DXL uses XML as a data unit, we use the SOAP approach based on HTTP. This means HTTP plays a role of transport protocol in the Web-based modeling and simulation. This makes Web browsers and Web servers the nucleus transferring messages. DXL models interacting with the message transferring units can therefore preserve the independence among programming languages. Since each DXL can be translated into different program languages, our Web-based modeling and simulation approach supports heterogeneous models and programming languages.

Figure 6-1. Web Architecture and DXL Approach for Web-based Modeling and Simulation

Integration of this Web-based modeling and simulation environment and XML data messaging, results in a new methodology for distributed modeling and simulation. Because each DXL block can operate on an XML input stream and produce an XML output, these blocks can become processes or models in a distributed system. A model itself can therefore be processed as a parameter of blocks in DXL so that multiple models can communicate with another model using model-based message passing. Such a
parameterization and distribution property of a model becomes an important factor in the
dynamic modification and extensibility of the model based on the composability and
reusability of model components. This can expand current modeling and simulation fields
to remote distributed modeling and simulation. We can also create a more complex and
better executed modeling and simulation system with low cost.

But, because of the unpredictability of current web environment, there is a need for
accurate timing issue for web-based distributed simulation. This problem limits current
web-based modeling and simulation to a distributed interactive simulation or an
educational application.
from array import array

MAX_TOKENS = 1000
MAX_SERVERS = 100
MAX_FACILITIES = 100
MAX_NUM_ATTR = 5
HEAP_SIZE = 50000
CALQSPACE = 49153
MAXNBUCKETS = 32768
UNIX = 1
UNIXX = 0
TURBOC = 0
NIL = 0
FREE = 0
BUSY = 1
ON = 1
OFF = 0
FOUND = 1
NOT_FOUND = -1
TIME_KEY = 0
AHEAD_PRIORITY_KEY = 1
BEHIND_PRIORITY_KEY = 2
INTERACTIVE = 0
BATCH = 1
LINKED = 0
HEAP = 1
CALENDAR = 2
HENRIK = 3
INOMIAL = 4
LEFTIST = 5
PAGODA = 6
PAIR = 7
SKEWDN = 8
SKEWUP = 9
SPLAY = 10
TWOLIST = 11
REMOVE_DUPLICATES = 16
RESEED = 32
true_ = 1
false_ = 0
SYNC = 0
ASYNC = 1

class TOKEN:
    def __init__(self):
        self.attr = array('d')
class NODE:
    def __init__(self):
        self.item = ITEM()
        self.next = None

class LIST_:
    def __init__(self):
        self.front = NODE()
        self.size = 0

class tokenstruct:
    event = 0
    time = 0.0
    first_arg = 0.0
    second_arg = 0.0

class ITEM:
    def __init__(self):
        self.time = 0.0
        self.event = 0
        self.priority = 0
        self.token = TOKEN()

class FACILITY:
    queue = None
    status = 0
    name = ""
    total_servers = 0
    busy_servers = 0
    total_busy_time = 0.0
    start_busy_time = 0.0
    preemptions = 0
    server_info = None

    def __init__(self):
        self.queue = LIST()
        self.server_info = []

class FACILITY_STATE:
    queue = ITEM();
    queue_length= 0
    status = 0
    name = ""
    total_servers = 0;
    busy_servers = 0;
    total_busy_time = 0.0;
    start_busy_time = 0.0;
    preemptions = 0;
    server_info = []

    def __init__(self):
        server_info = array('i', array('i'))
class STATE:
    time = 0.0
    future_events = []
    future_events_length = 0
    facility = []
    facility_length = 0
    heap_count = 0
    heap = []

def __init__(self):
    future_events = ITEM()
    facility = FACILITY_STATE()
    heap = ITEM()

token_list = []
current_time = last_event_time = total_token_time = 0.0
current_event = facilities = arrivals = completions = 0
tokens_in_system = trace_flag = trace_type = heap_count = 0
event_list_type = remove_duplicates = 0
current_operation = ""
event_list = LIST_()
heap = []
facility = []

utilization = idle = arrival_rate = throughput = total_sim_time = 0.0
total_busy_time = total_utilization = 0.0
mean_service_time = mean_num_tokens = mean_residence_time = 0.0

strm = 1
rn_stream = 0

def init(set_time, flags):
    i = 0
    type = 0
    ranmark = 0

    type = flags & 15
    ranmark = flags & 32
    remove_duplicates = flags & 16

    for i in xrange(0, MAX_TOKENS):
        token_list.append(tokenstruct())
    for i in xrange(0, HEAP_SIZE):
        heap.append(ITEM())
    for i in xrange(0, MAX_FACILITIES):
        facility.append(FACILITY())
    for i in xrange(0, MAX_TOKENS):
        token_list[i].event = 99999
    event_list_type = type
    create_list(event_list)
    heap_count = 0
    trace_flag = OFF
    current_time = set_time
    last_event_time = set_time
    facilities = 0
    arrivals = 0
completions = 0
total_token_time = 0
tokens_in_system = 0
calresize_enable = true_
rn_stream += 1

if rn_stream > 15:
    rn_stream = 1

def create_list(tmp_list):
    tmp_list.front = None
    tmp_list.size = 0

def schedule(event, inter_time, token):
    i=0
    an_item = ITEM()
    token_id = token.attr[0] % MAX_TOKENS
    event_time = current_time + inter_time
    if ((token_list[token_id].event != event) or
        (token_list[token_id].time != event_time) or
        (token_list[token_id].first_arg != token.attr[1]) or
        (token_list[token_id].second_arg != token.attr[2])):
        token_list[token_id].event = event
        token_list[token_id].time = event_time
        token_list[token_id].first_arg = token.attr[1]
        token_list[token_id].second_arg = token.attr[2]
    an_item.time = event_time
    an_item.event = event
    for i in xrange(0, MAX_NUM_ATTR):
        an_item.token_attr[i] = token.attr[i]

    if (event_list_type == LINKED):
        insert_list(event_list, an_item, TIME_KEY)
    elif (event_list_type == HEAP):
        heap_insert(an_item)

def heap_insert(tmp_item):
    parent = child = 0
    item = ITEM()
    
    ITEM_copy(item, tmp_item)
    heap_count+=1
    heap[heap_count] = item
    if(heap_count > 1):
        child = heap_count
        parent = child/2
        while ((heap[parent].time > heap[child].time) and (child > 1)):
            heap_swap(heap[parent],heap[child])
        child = parent
        if (child > 1):
            parent = child/2
def heap_swap(item1, item2):
    temp = ITEM()
    ITEM_copy(temp, item1)
    ITEM_copy(item1, item2)
    ITEM_copy(item2, temp)

def heap_delete(token_id, tmp_item_addr):
    i = j = parent = child = 0
    i = 1
    while (heap[i].token.attr[0] != token_id):
        i+=1
    heap_swap(heap[i],heap[heap_count])
    ITEM_copy(tmp_item_addr, heap[heap_count])
    heap_count-=1

    heap_up_adjust(i)
    heap_down_adjust(i)

def heap_remove(tmp_item_addr):
    parent = child = 0
    if (tmp_item_addr != None and heap_count > 0):
        ITEM_copy(tmp_item_addr, heap[1])
        heap_swap(heap[1],heap[heap_count])
        heap_count-=1
        return 0
    else:
        return -1

def heap_read(tmp_item_addr):
    if (tmp_item_addr != None and heap_count > 0):
        ITEM_copy(tmp_item_addr, heap[1])
        return 0
    else:
        return -1

def heap_up_adjust(child):
    parent = child/2
    while (parent>0):
        if (heap[parent].time < heap[child].time):
            break
        heap_swap(heap[parent], heap[child])
        child = parent
        parent /= 2

def heap_down_adjust(parent):
    child = 0
    temp_true = 1
    while (temp_true==1):
        if (2*parent > heap_count):
            temp_true = 0
            break
        else:
            child = 2*parent
if (child+1 <= heap_count):
    if (heap[child+1].time < heap[child].time):
        child+=1

if (heap[parent].time < heap[child].time):
    temp_true = 0
    break

heap_swap(heap[parent], heap[child])
parent = child

def print_heap():
    i=0
    for i in xrange(1, heap_count+1):
        print(heap[i].time + " ")
    print()

def create_facility(tmp_name, num_servers):
    i = 0

    facilities++=1
    create_list(facility[facilities].queue)
    facility[facilities].status = FREE
    facility[facilities].name = String(tmp_name)
    facility[facilities].total_servers = num_servers
    facility[facilities].busy_servers = 0

    for i in xrange(1, num_servers+1):
        facility[facilities].server_info[i][0] = 0
        facility[facilities].server_info[i][1] = 0

    facility[facilities].preemptions = 0
    facility[facilities].total_busy_time = 0.0
    return(facilities)

def next_event(tmp_token_ptr, ptime, mode):
    i = status = 0
    vtime = 0.0
    an_item = ITEM()
    token_ptr = TokenPointer()

    if(event_list_type == LINKED):
        status=list_read(event_list,an_item)
        if (status == -1):
            return -1
        elif(event_list_type == HEAP):
            status=heap_read(an_item)
            if (status == -1):
                return -1

    vtime = an_item.time

    if(mode==ASYNC or (mode==SYNC and ptime>=vtime)):
return -1
elif(event_list_type == HEAP):
    status = heap_remove(an_item)
    if (status == -1):
        return -1

    current_time = an_item.time
    current_event = an_item.event

    for i in xrange(MAX_NUM_ATTR):
        tmp_token_ptr.attr[i] = an_item.token.attr[i]

    total_token_time += tokens_in_system*(time() - last_event_time)
    last_event_time = time()

    return(an_item.event)
else:
    return -1

def cancel_event(event):
    current_node = None
    previous_node = None
    temp_ptr = None
    list_ptr = None

    list_ptr = event_list

    current_node = list_ptr.front
    previous_node = list_ptr.front

    while((current_node != None) and (current_node.item.event != event)):
        previous_node = current_node
        current_node = current_node.next

    if(current_node == None):
        return(NOT_FOUND)
    else:
        if(previous_node == current_node):
            temp_ptr = list_ptr.front
            list_ptr.front = current_node.next
        else:
            temp_ptr = current_node
            previous_node.next = current_node.next

        list_ptr.size-=1
        return (current_node.item.token.attr[0])

def cancel_token(token):
    current_node = None
    previous_node = None
    temp_ptr = None
    list_ptr = None

    list_ptr = event_list
    current_node = list_ptr.front
    previous_node = list_ptr.front
while((current_node!=None) and (current_node.item.token.attr[0] !=
token.attr[0])):
    previous_node = current_node
    current_node = current_node.next

if(current_node==None):
    return(NOT_FOUND)
else:
    if(previous_node==current_node):
        temp_ptr = list_ptr.front
        list_ptr.front = current_node.next
    else:
        temp_ptr = current_node
        previous_node.next = current_node.next

list_ptr.size+=1
return (current_node.item.event)

def request(facility_id, token, priority):
an_item = ITEM()
i = server_num = 0

if (facility[facility_id].busy_servers == 0):
    facility[facility_id].start_busy_time = time()

if (facility[facility_id].status == FREE):
    facility[facility_id].busy_servers+=1
    server_num = 1
    while (facility[facility_id].server_info[server_num][0] != 0):
        server_num+=1
    facility[facility_id].server_info[server_num][0] = token.attr[0]
    facility[facility_id].server_info[server_num][1] = priority
    if (facility[facility_id].busy_servers ==
facility[facility_id].total_servers):
        facility[facility_id].status = BUSY
        return(FREE)
else:
an_item.time = current_time
for i in xrange(MAX_NUM_ATTR):
an_item.token.attr[i] = token.attr[i]
an_item.priority = priority
an_item.event = current_event
insert_list(facility[facility_id].queue, an_item,
BEHIND_PRIORITY_KEY)
return(BUSY)

def insert_list(list_ptr, tmp_item_ptr, key):
current_node = None
previous_node = None
new_node = NODE()

ITEM_copy(new_node.item, tmp_item_ptr)
new_node.next = None
list_ptr.size += 1
current_node = list_ptr.front
previous_node = list_ptr.front

if (key == TIME_KEY):
    while ((current_node != None) and (current_node.item.time <=
new_node.item.time)):
        previous_node = current_node
        current_node = current_node.next

elif (key == BEHIND_PRIORITY_KEY):
    while ((current_node != None) and (current_node.item.priority
>= new_node.item.priority)):
        previous_node = current_node
        current_node = current_node.next

else:
    while ((current_node != None) and (current_node.item.priority >
new_node.item.priority)):
        previous_node = current_node
        current_node = current_node.next

if (current_node == None):
    if (previous_node == None):
        list_ptr.front = new_node
    else:
        previous_node.next = new_node
else:
    if (previous_node != current_node):
        previous_node.next = new_node
        new_node.next = current_node
    else:
        list_ptr.front = new_node
        new_node.next = current_node

def preempt(facility_id, token, priority):
    an_item = ITEM()
    heap_item = ITEM()

    server_num = do_preempt = i = preempted_token = 0
    num_servers = minimum_priority = server_with_min = 0
    preempted_token_priority = 0

    if (facility[facility_id].busy_servers == 0):
        facility[facility_id].start_busy_time = time()

    if (facility[facility_id].status == FREE):
        facility[facility_id].start_busy_time = time()

    if (facility[facility_id].status == FREE):
        facility[facility_id].busy_servers+=1

        server_num = 1
        while (facility[facility_id].server_info[server_num][0] != 0):
            server_num+=1
            facility[facility_id].server_info[server_num][0] = token.attr[0]
            facility[facility_id].server_info[server_num][1] = priority
            if (facility[facility_id].busy_servers ==
facility[facility_id].total_servers):
facilities[facility_id].status = C_BUSY
    return(FREE)
else:
    minimum_priority = 9999
    num_servers = facilities[facility_id].total_servers

    for i in xrange(1, num_servers+1):
        if (facilities[facility_id].server_info[i][1] <
            minimum_priority):
            minimum_priority =
            facilities[facility_id].server_info[i][1]
            server_with_min = i

        if (priority > minimum_priority):
            do_preempt = true_
        else:
            do_preempt = false_

        if (do_preempt == true_):
            facilities[facility_id].preemptions+=1
            preempted_token =
            facilities[facility_id].server_info[server_with_min][0]

            preempted_token_priority =
            facilities[facility_id].server_info[server_with_min][1]
            facilities[facility_id].server_info[server_with_min][0] =
            token.attr[0]
            facilities[facility_id].server_info[server_with_min][1] =
            priority

            if(event_list_type==LINKED):
                an_item = listrmqueue(preempted_token,event_list)
                an_item.priority = preempted_token_priority
                an_item.time -= time()
                an_item.time = -an_item.time
                insert_list(facilities[facility_id].queue,an_item,AHEAD_PRIORITY_KEY)
            elif(event_list_type==HEAP):
                heap_delete(preempted_token,heap_item)
                an_item.event = heap_item.event
                an_item.priority = preempted_token_priority
                an_item.time = heap_item.time - time()
                an_item.time = -an_item.time
                an_item.token = heap_item.token
                insert_list(facilities[facility_id].queue,an_item,AHEAD_PRIORITY_KEY)
            return(FREE)
        else:
            an_item.time = current_time
            for i in xrange(MAX_NUM_ATTR):
                an_item.token.attr[i] = token.attr[i]
                an_item.priority = priority
                an_item.event = current_event
            insert_list(facilities[facility_id].queue,an_item,BEHIND_PRIORITY_KEY)
    return(BUSY)
def listrmqueue(n, tmp_list_ptr):
    list_ptr = ListPointer()
    list_ptr.ptr = tmp_list_ptr
    temp = ITEM()
    current_node = NodePointer()
    previous_node = NodePointer()
    temp_ptr = NodePointer()
    current_node.ptr = list_ptr.ptr.front.ptr
    previous_node.ptr = list_ptr.ptr.front.ptr
    while ((current_node != None) and
           (current_node.ptr.item.token.attr[0] != n)):
        previous_node.ptr = current_node.ptr
        current_node.ptr = current_node.ptr.next.ptr
    if(current_node == None):
        print("PREEMPT : Attempt to preempt a non-existent token<P")
        print("Token # " + n)
        return temp
    else:
        temp = current_node.ptr.item
        if (previous_node.ptr == current_node.ptr):
            temp_ptr.ptr = list_ptr.ptr.front.ptr
            list_ptr.ptr.front.ptr = current_node.ptr.next.ptr
        else:
            temp_ptr.ptr = current_node.ptr
            previous_node.ptr.next.ptr = current_node.ptr.next.ptr
        list_ptr.ptr.size-=1
        return temp

def release(facility_id, token):
    an_item = ITEM()
    server_num = i = found = 0
    server_num = facility[facility_id].total_servers
    found = false_
    i = 1
    while((found !=0) and i<=server_num):
        if(facility[facility_id].server_info[i][0] == token.attr[0]):
            facility[facility_id].server_info[i][0] = 0
            facility[facility_id].server_info[i][1] = 0
            found = true_
            i+=1
        if(found==0):
            facility[facility_id].status = FREE
            facility[facility_id].busy_servers-=1
            if(facility[facility_id].busy_servers == 0):
                facility[facility_id].total_busy_time += time() -
                facility[facility_id].start_busy_time
                facility[facility_id].start_busy_time = time()
            if(facility[facility_id].queue.size > 0):
                remove_front_list(facility[facility_id].queue,an_item)
if (an_item.time < 0):
    an_item.time = current_time - an_item.time

if(event_list_type==LINKED):
    insert_list(event_list, an_item, TIME_KEY)
elif(event_list_type==HEAP):
    heap_insert(an_item)

facility[facility_id].status = BUSY
facility[facility_id].busy_servers+=1

server_num = 1
while (facility[facility_id].server_info[server_num][0] != 0):
    server_num+=1
    facility[facility_id].server_info[server_num][0] = an_item.token.attr[0]
    facility[facility_id].server_info[server_num][1] = an_item.priority
else:
    an_item.time = current_time
    if(event_list_type==LINKED):
        add_front_list(event_list, an_item)
elif(event_list_type==HEAP):
        heap_insert(an_item)

def add_front_list(tmp_list_ptr, tmp_item_ptr):
    list_ptr = ListPointer()
    item_ptr = ItemPointer()
    list_ptr.ptr = tmp_list_ptr
    item_ptr.ptr = tmp_item_ptr

    new_node = NodePointer()
    new_node.ptr = NODE()

    ITEM_copy(new_node.ptr.item, item_ptr.ptr)
    new_node.ptr.next.ptr = None
    if (list_ptr.ptr.size == 0):
        list_ptr.ptr.front.ptr = new_node.ptr
    else:
        new_node.ptr.next.ptr = list_ptr.ptr.front.ptr
        list_ptr.ptr.front.ptr = new_node.ptr

    list_ptr.ptr.size += 1

def remove_front_list(tmp_list_ptr, tmp_item_ptr):
    if (tmp_list_ptr != None and tmp_list_ptr.front.ptr != None):
        temp_ptr = NodePointer()
        ITEM_copy(tmp_item_ptr, tmp_list_ptr.front.ptr.item)
        temp_ptr.ptr = tmp_list_ptr.front.ptr
        tmp_list_ptr.front.ptr = tmp_list_ptr.front.ptr.next.ptr
        tmp_list_ptr.size -= 1
        return 0
    else:
        return -1

def list_read(tmp_list_ptr, tmp_item_ptr):
if (tmp_list_ptr != None and tmp_list_ptr.front.ptr != None):
    ITEM_copy(tmp_item_ptr, tmp_list_ptr.front.ptr.item)
    return 0
else:
    return -1

def trace_facility(facility_id):
    node = NodePointer()
    list_ptr = ListPointer()
    i = 0
    list_ptr.ptr = facility[facility_id].queue
    print("Time: " + time())
    print("Queue " + facility_id + ": ")
    node.ptr = list_ptr.ptr.front.ptr
    while (node.ptr != None):
        print("(TM " + node.ptr.item.time + ")")
        print("EV " + node.ptr.item.event + ")")
        print("TK " + node.ptr.item.token.attr[0] + ")")
        node.ptr = node.ptr.next.ptr
    print()
    for i in xrange(60):
        print("-"
    print()

def trace_eventlist():
    node = NodePointer()
    list_ptr = ListPointer()
    i = 0
    list_ptr.ptr = event_list
    print("Time: " + time())
    print("Events: ")
    node.ptr = list_ptr.ptr.front.ptr
    while (node.ptr != None):
        print("(TM " + node.ptr.item.time + ")")
        print("EV " + node.ptr.item.event + ")")
        print("TK " + node.ptr.item.token.attr[0] + ")")
        node.ptr = node.ptr.next.ptr
    print()
    for i in xrange(60):
        print("-"
    print()

def state():
    list_ptr = ListPointer()
    node = NodePointer()
    i = j = 0
    any_char = ""
    if (event_list_type==LINKED):
        list_ptr.ptr = event_list
        node.ptr = list_ptr.ptr.front.ptr
        i = 0
        while (node.ptr != None):
            ...
state_tmp.future_events[i] = ITEM()
ITEM_copy(state_tmp.future_events[i], node.ptr.item)
i+=1
node.ptr = node.ptr.next.ptr

state_tmp.future_events_length = i

else:
    state_tmp.heap_count = heap_count
    for i in xrange(heap_count):
        state_tmp.heap[i] = ITEM()
        ITEM_copy(state_tmp.heap[i], heap[i])

state_tmp.facility_length = facilities
for i in xrange(1, facilities+1):
    state_tmp.facility[i] = FACILITY_STATE()
    state_tmp.facility[i].name = String(facility[i].name)
    state_tmp.facility[i].total_servers = facility[i].total_servers
    state_tmp.facility[i].busy_servers = facility[i].busy_servers

    for j in xrange(1, facility[i].total_servers+1):
        state_tmp.facility[i].server_info[j][0] =
        facility[i].server_info[j][0]
        state_tmp.facility[i].server_info[j][1] =
        facility[i].server_info[j][1]

    list_ptr.ptr = facility[i].queue
    node.ptr = list_ptr.ptr.front.ptr

    j=0
    while(node.ptr!=None):
        state_tmp.facility[i].queue[j] = ITEM()
        ITEM_copy(state_tmp.facility[i].queue[j], node ptr.item)
        j+=1
    node.ptr = node.ptr.next.ptr

state_tmp.facility[i].queue_length = j

return state_tmp

def report_stages():
    list_ptr = ListPointer()
    node = NodePointer()
    i = j = 0
    any_char = ''

    println("## TIME: " + time())
println("## " + current_operation)
println()

    if(event list_type==LINKED):
        println("## EVENT LIST")
        list_ptr.ptr = event_list
        println("")
node.ptr = list_ptr.ptr.front.ptr
while(node.ptr!=None):
    println(" ")
print("++++++")
node.ptr = node.ptr.next.ptr

print()
print("Token ")
node.ptr = list_ptr.ptr.front.ptr
while(node.ptr!=None):
    print(" ")
    print("| " + node.ptr.item.token.attr[0] + "|")
    node.ptr = node.ptr.next.ptr

print()
print("Time ")
node.ptr = list_ptr.ptr.front.ptr
while(node.ptr!=None):
    print("<=")
    print("|" + node.ptr.item.time + "|")
    node.ptr = node.ptr.next.ptr

print()
print("Event ")
node.ptr = list_ptr.ptr.front.ptr
while(node.ptr!=None):
    print(" ")
    print("| " + node.ptr.item.event + "|")
    node.ptr = node.ptr.next.ptr

print()

print(" ")
node.ptr = list_ptr.ptr.front.ptr
while(node.ptr!=None):
    print(" ")
    print("++++++")
    node.ptr = node.ptr.next.ptr

else:
    print("## PRIORITY QUEUE")
print_heap()
for i in xrange(1, facilities+1):
    print("## FACILITY " + i + ": (" + facility[i].name + ")", " + facility[i].total_servers + " Server(s), " + facility[i].busy_servers + " Busy.")
    print("Server(s): ")
    for j in xrange(1, facility[i].total_servers+1):
        print("(" + j + ") TK " + facility[i].server_info[j][0] + " PR " + facility[i].server_info[j][1])
    print()

list_ptr.ptr = facility[i].queue
print(" ")
node.ptr = list_ptr.ptr.front.ptr
while (node.ptr != None):
    print(" ")
    print("+++++")
    node.ptr = node.ptr.next.ptr

print()

print("Token ")
node.ptr = list_ptr.ptr.front.ptr
while (node.ptr != None):
    print(" ")
    print("|   " + node.ptr.item.token.attr[0] + "|")
    node.ptr = node.ptr.next.ptr

print()

print("Time ")
node.ptr = list_ptr.ptr.front.ptr
while (node.ptr != None):
    print("<=")
    print("|" + node.ptr.item.time + "|")
    node.ptr = node.ptr.next.ptr

print()

print("Event")
node.ptr = list_ptr.ptr.front.ptr
while (node.ptr != None):
    print(" ")
    print("|   " + node.ptr.item.event + "|")
    node.ptr = node.ptr.next.ptr

print()

print("Priority")
node.ptr = list_ptr.ptr.front.ptr
while (node.ptr != None):
    print(" ")
    print("|   " + node.ptr.item.priority + "|")
    node.ptr = node.ptr.next.ptr

print()

print(" ")
node.ptr = list_ptr.ptr.front.ptr
while (node.ptr != None):
    print(" ")
    print("+++++")
    node.ptr = node.ptr.next.ptr

print()

def time():
    return (current_time)

def busy_time(facility_id):
    return (facility[facility_id].total_busy_time)
def update_arrivals():
    arrivals += 1
    tokens_in_system += 1

def update_completions():
    completions += 1
    tokens_in_system -= 1

def report_stats():
    i = 0

    println("+---------------------------------+")
    println("| SimPackJS SIMULATION REPORT    |")
    println("+---------------------------------+")

    if completions == 0:
        completions = 1
    total_sim_time = time()
    total_busy_time = 0.0
    for i in xrange(1, facilities + 1):
        if facility[i].busy_servers > 0:
            facility[i].total_busy_time += time() -
            facility[i].start_busy_time
    total_busy_time += busy_time(i)

    total_busy_time /= facilities
    total_utilization = total_busy_time / total_sim_time
    arrival_rate = arrivals / total_sim_time
    throughput = completions / total_sim_time
    mean_service_time = total_busy_time / completions
    mean_num_tokens = total_token_time / total_sim_time
    mean_residence_time = mean_num_tokens / throughput

    print("Total Simulation Time: "+total_sim_time)
    print("Total System Arrivals: "+arrivals)
    print("Total System Completions: "+completions)

    print("System Wide Statistics")
    print("---------------------")
    print("System Utilization: "+100.0*total_utilization)
    print("Arrival Rate: "+arrival_rate+" Throughput: "+throughput)
    print("Mean Service Time per Token: "+mean_service_time)
    print("Mean # of Tokens in System: "+mean_num_tokens)
    print("Mean Residence Time for each Token: "+mean_residence_time)

    print("Facility Statistics")
    print("---------------------")
    for i in xrange(1, facilities + 1):
        utilization = 100.0 * busy_time(i) / time()
        idle = 100.0 * (time() - busy_time(i)) / time()
        print("F "+i+" ("+facility[i].name+") : Idle: "+idle+", Util: "+utilization+", Preemptions: "+facility[i].preemptions)

    def facility_size(facility_id):
        return(facility[facility_id].queue.size)
def ITEM_copy(target, source):
    target.time = source.time
    target.event = source.event
    TOKEN_copy(target.token, source.token)
    target.priority = source.priority

def TOKEN_copy(target, source):
    for i in xrange(MAX_NUM_ATTR):
        target.attr[i] = source.attr[i]
LIST OF REFERENCES


64. SODIPODI. February 1, 2005. http://sodipodi.sourceforge.net/


BIOGRAPHICAL SKETCH

Jinho Lee received his Bachelor of Science degree in information engineering from Sungkyunkwan University, Seoul, Korea, in 1997. In 1999, he received his Master of Science degree in electrical computer engineering from Sungkyunkwan University (Seoul, Korea). After his graduation, he worked as a lecturer at Doowon Technical College and Kaywon School of Arts & Design. He began working toward his PhD degree in 2000. His major research areas are XML-based multimodeling and simulation, web-based modeling and simulation, and code generation.