**In compliance with the
Canadian Privacy Legislation
some supporting forms
may have been removed from
this dissertation.**

**While these forms may be included
in the document page count,
their removal does not represent
any loss of content from the dissertation.**

Integrating Timing Diagram Protocols with HLA Simulations

By

Hossam Khalil

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment
of the requirements for the degree of

M.Sc in Information and System Science (ISS)

**School of Computer Science
Carleton University, Ottawa, Ontario**

May 2003

National Library
of Canada

Bibliothèque nationale
du Canada

Acquisitions and
Bibliographic Services

Acquisisitons et
services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 0-612-88773-1

Our file Notre référence

ISBN: 0-612-88773-1

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

Canada

The undersigned hereby recommends to the Faculty of Graduate Studies and Research
the acceptance of the thesis:

Integrating Timing Diagram Protocols with HLA Simulations

Submitted By

Hossam Khalil

In partial fulfillment of the requirements for the degree of M.Sc in Information and
System Science (ISS)

Professor Trevor Pearce, Thesis Supervisor

Director, School of Computer Science

Carleton University
May 2003

Abstract

Manufacturers and integrators of hardware components and Systems-on-Chip Intellectual Property need to model and simulate the hardware interfacing behaviors of the product before the physical integration into a hardware system. This thesis focuses on the simulation of hardware systems interfacing behaviors based on the system components' signaling scenarios in the modeling process. Timing diagrams are employed to specify the signaling scenarios since timing diagrams have a wide acceptance among hardware professionals in describing, visualizing and analyzing signaling behavior and timing relations. The High Level Architecture (HLA) is used as a simulation platform since the HLA has gained rapid popularity for supporting simulation interoperability and reusability. The thesis proposes a simulator architecture and design, which follows the object-oriented paradigm and satisfies simulation interoperability goals. The proposed architecture reuses component simulations, which interact at levels of abstraction above the details of signaling, and introduces new modules that encapsulate and enable signaling behaviors.

Keywords: *modeling and simulation, hardware interfacing behavior, timing diagrams specification, Timing Diagram Markup Language (TDML), High Level Architecture (HLA), Object Oriented architecture and design, interoperability, reusability and scalability.*

Acknowledgements

A thesis cannot be written in isolation, and there are many people whom I would like to thank for their assistance during the course of my research.

First, I would like to thank my supervisor Dr. Trevor Pearce for his patience and support. Although I frequently resisted his suggestions (and usually I was wrong), this thesis has benefited greatly due to his advice. Sometimes he offered me hot coffee.

Thanks to my colleagues who have contributed to my research by providing advices and remarkable comments. For this, I would like to extend a thank you to Fayyaz Ahmad, Hui Shen, Yi Lei (I never follow their comments and usually they were correct). Thanks very much to Zhang Lin who criticized the last 3000 graphs and slides I have made. (Most of them were redrawn or canceled)

Thanks to my children, Marwan and Aya, who have never known their father but a student. (I can never work at home unless they are asleep). Thanks to Eman Abou El-Ela for her great support. (I have not seen her for a while)

Very special thanks to my mother Ibtesam Sharaf El-Din and my father Sayed Ahmed Khalil for everything.

Table of Contents

Abstract	iii
List of Figures	viii
List of Tables	xi
Chapter 1: Introduction	1
Chapter 2: Background	5
2-1 Timing Diagrams and Waveforms	6
2-2 Timing Diagram Markup Language (TDML)	7
2-3 High Level Architecture (HLA)	8
2-4 Simulation Reusability	10
2-5 Related Studies	11
Chapter 3: HLA Compliant System Simulator	14
3-1 Motivation: a Tool for Modeling	15
3-2 Research Questions	16
3-3 Problem Definition	17
3-4 Scope	19
3-5 Research Contributions	20
3-6 Solution	21
3-6-1 Evolving Existing Models	23
3-6-2 Write Scenario Example	27
3-6-3 Methodology and Support Tool	31
Chapter 4: Phase I: Building a Protocol Model	38

4-1 Events Segment	42
4-2 Component-Signal Relation Segment	46
4-3 AC-Characteristics Segment	50
4-4 Flags Segment	55
4-5 Protocol States Segment	61
4-6 Dataflow Segment	67
Chapter 5: Phase II: System Model	79
5-1 System Model Construction	81
5-2 System Model Elements	83
5-2-1 Simulator Element	85
5-2-2 Agent Element	85
5-2-3 Command Element	87
5-3 Wrapping the System Model	94
5-4 Tool Specifications of Creating a System Model	95
5-5 System Model Dataflow	96
Chapter 6: Phase III HLA Harness	99
6-1 Harness Message Passing (MP) Mechanism	101
6-1-1 Building the MP Manager	102
6-1-2 MP Dynamics Example	105
6-2 Service Manager	107
6-2-1 Production Controller	109
6-2-2 Consumption Controller	109
6-2-3 Progress Controller	110

6-2-4 Two-Federate Approach to Time Advance	110
6-3 Runtime Scenarios	113
Chapter 7: Conclusions	121
7-1 Conclusions	121
7-2 Future Research	123
7-2-1 Dynamic Code Generation	123
7-2-2 Multi Layer GUI for Protocol Model Data Collection	124
7-2-3 Introducing a Migration Methodology from a non-HLA Simulation to an HLA Simulation	125
References	126

List of Figures

Figure 1-1	Three Phases Modeling Tower	3
Figure 2-1	Systems-On-a-Chip Intellectual Property Business Model	7
Figure 2-2	Automatic Generation of Bus Interface Models by Using Timing Diagrams and State Diagram as Inputs	13
Figure 3-1	HLA Compliant Simulator Architecture	23
Figure 3-2	Target Atomic Events of Existing Simulation	24
Figure 3-3	Messages Sequence Chart for Write/Read	25
Figure 3-4	Event Propagation	27
Figure 3-5	Write Scenario Timing Specification	29
Figure 3-6	Sample Example of Protocol Interactions	30
Figure 3-7	Constructing a Protocol Model	34
Figure 3-8	The Construction of the Communication Media Harness	36
Figure 3-9	Publisher/Subscriber Message Sequence Chart	37
Figure 4-1	Protocol Model Construction Process	39
Figure 4-2	Data Collection Step	40
Figure 4-3	Protocol Model Generation Step	41
Figure 4-4	Events Segment Data Collection Step	42
Figure 4-5	Simplified Read Behavior Waveforms	43
Figure 4-6	Timing Tool Simplified Read Behavior Visualization	44
Figure 4-7	Events Segment Corresponding Data Structures	45
Figure 4-8	Diagram Package Structure	46
Figure 4-9	Presentation Layer of Component-Signal Relation	48

Figure 4-10	Component-Signal Relation XML Structure	49
Figure 4-11	AC-Characteristic Presentation Layer	52
Figure 4-12	AC-Characteristics XML Sample	53
Figure 4-13	AC-Characteristic Package in An UML Diagram	55
Figure 4-14	Flags Presentation Layer	58
Figure 4-15	Sample from Flags Segment XML File of a Read Protocol Example	60
Figure 4-16	Flag Package Structure	61
Figure 4-17	Protocol States XML Main Structure Sample from Read Example	63
Figure 4-18	XML Structure for Next Protocol State Sample from Read Example	65
Figure 4-19	Protocol State Package Part A	66
Figure 4-20	Protocol State Package Part B	67
Figure 4-21	Read Protocol Signature	68
Figure 4-22	Protocol Variable Data Path	71
Figure 4-23	Data Collection of Dataflow Segment	72
Figure 4-24	Read Signature XML Structure	73
Figure 4-25	<Data_Event> XML Element	74
Figure 4-26	Address Path XML Structure in Read	75
Figure 4-27	Dataflow Presentation Layer	77
Figure 4-28	Dataflow XML Inputs in Protocol Model Generation	78
Figure 4-29	Dataflow Package UML Class Diagram	78

Figure 5-1	The System Model in System Simulator Architecture	81
Figure 5-2	Constructing a System Model	82
Figure 5-3	System Model Factory	83
Figure 5-4	System Model Elements	83
Figure 5-5	System Model Simplified UML Class Diagram	84
Figure 5-6	Example of Agent Tree UML Class Diagram	87
Figure 5-7	Example of Command Tree UML Class Diagram	89
Figure 5-8	The run Method in the Command Thread	90
Figure 5-9	Event Processing	91
Figure 5-10	Updating Protocol Flags by a Current Event	92
Figure 5-11	The Processing of a Data Event	93
Figure 5-12	Tool Specifications for Generating the System Model	96
Figure 5-13	System Model Dataflow Example	98
Figure 6-1	HLA Harness Architecture	100
Figure 6-2	MP Manager Design Elements	104
Figure 6-3	HLA-MP Mechanism	107
Figure 6-4	HLA Service Manager	108
Figure 6-5	Producing and Consuming TSO Events	113
Figure 6-6	Only One Federate Produces an Event	115
Figure 6-7	Each Federate Produces an Event	117
Figure 6-8	No Production of Events	118
Figure 6-9	Deadlock Scenario	119
Figure 6-10	Advancing Time in a Federation of more than Two Federates	120

List of Tables

Table 3-1	Write Atomic Event	25
-----------	--------------------	----

Chapter 1: Introduction

This thesis introduces a methodology and a support tool for the modeling and simulation of hardware systems. The approach enables the generic creation of a High Level Architecture (HLA) compliant simulator for systems that interact using signaling behaviors specified in timing diagrams. The methodology assumes that a course-grained simulation of the target system already exists, in which the target system components interact using abstract atomic events. The atomic events encompass sequences of fine-grained signaling behaviors. The methodology facilitates the reuse and refinement of the existing simulation to study the system's behaviors at the granularity of signal interactions. The High Level Architecture (HLA) is explicitly emphasized as the simulation platform.

The proposed tool and methodology are useful in the market of hardware components and Systems-on-Chip (SoC) Intellectual Property (IP). Providers and customers need model to test products before physical integration [8]. The research facilitates extending existing models, which discount system signaling scenarios. Evolving existing models is an advantage, because it enables the reusability of other verified work and provides an extended functionality for the reused models. Using timing diagrams in specifying signaling behaviors is another modeling advantage because of its convenience. Timing diagrams are commonly used in the hardware industry, to visualize and analyze timing relations among system signals [19]. Moreover, the recent introduction of the Timing Diagram Markup Language facilitates the exchange of timing diagrams in a standard format [8]. The decision to target the HLA simulation platform adds various benefits to

the model. The HLA is the mandatory simulation platform of the defense community [1], and it allows simulation interoperability and reusability [4].

The proposed methodology in this thesis defines a framework for constructing HLA compatible simulations of hardware models. HLA compatibility entitles the model to all of the advantages of the HLA simulation platform, and makes it competitive in HLA-based simulation contracts. The modeling architecture and design follows an object-oriented approach to provide a scalable and reusable simulation model. The model can be easily maintained, updated, extended and reused [28]. Finally, this study highlights a methodology for simulation migration from a non-HLA simulation to an HLA compatible simulation. This approach will enable the reuse of existing non-HLA simulations wherever the HLA platform is mandated.

The HLA compliant simulator is constructed in three consecutive building phases. As shown in figure 1-1, the first phase models each single behavior scenario in a protocol model using its timing diagram specifications. The second phase integrates all protocol models to produce a system model, while the third phase utilizes the HLA Runtime Infrastructure (RTI) simulation platform allowing component federates to simulate the system model.

Figure 1-1: Three Phases Modeling Tower

In this dissertation, chapter 2 highlights topics related to the construction of HLA simulators of systems specified using timing diagrams. It demonstrates the use of timing diagrams in specifying signaling behaviors. It also introduces the Timing Diagram Markup Language (TDML), which enables the exchange of timing diagrams in an XML-based format. Chapter 2 also highlights the historical background of the HLA simulation platform and concludes by reviewing some work related to the thesis research.

Chapter 3 summarizes the research. It begins with the motivation of the research illustrating its expected benefits in the market of hardware systems and simulations. The motivation is followed by the central thesis questions and problem definition. The thesis scope and the research approach are also identified. Chapter 3 then introduces the proposed solution of constructing an HLA system simulator, and concludes with the contributions of the research.

The solution is further detailed in chapters 4, 5 and 6. Chapter 4 introduces phase I of the construction of an HLA-compliant system simulator. It shows how each individual system behavior is described in a protocol model. Chapter 5 explains phase II of the simulator construction. It illustrates the integration process of all phase I protocol models to create a system model wrapper. It introduces component interactions through a messaging passing mechanism, which is contracted by a Communication Interface. Chapter 6 illustrates phase III of constructing the simulator. It shows how to define an HLA framework to simulate the system model by creating an HLA harness. The chapter explains the role of the HLA harness in communicating messages across the HLA platform. Chapter 7 states the research conclusions and provides future work suggestions.

Chapter 2: Background

This chapter highlights the major topics related to the construction of an HLA compliant simulator of system behaviors specified in timing diagrams. The chapter begins with a discussion of the role of timing diagrams as the timing specification measures that are accepted widely among electronic components' specialists and the Intellectual Property market.

The second section introduces the Timing Diagram Markup Language (TDML), which represents timing diagrams in an exchangeable format. TDML, as an XML-based language, supports interoperability requirements of simulations in a heterogeneous distributed computing environment [15, 21].

The third section highlights the High Level Architecture (HLA) simulation platform which is mandated by the USA Department of Defense (DoD) for all of its modeling and simulation work [4]. The HLA was born under the necessity of decreasing budgets. Its solution is to support the reusability [3, 18] and the interoperability of simulation components [14].

The fourth section highlights simulation reusability. The section shows fact sheets of simulation cost that highlight the importance of simulation reusability. The worth and cost of a simulation model are examined in terms of its life span and its amount of reuse [6]. Section five overviews several other related studies.

Subsequent chapters propose and demonstrate a framework for constructing an HLA simulator for hardware components. The model describes the behaviors of the system components at the level of signal interactions. The signaling scenarios of system behaviors are described in timing diagrams. The intent of the thesis is to introduce an extensible reusable solution, which exercises modeling and separation of concerns in its design.

2-1 Timing Diagrams and Waveforms

A waveform is simply the representation of a signal as a plot of amplitude versus time [12]. In other words, a waveform is a graphic representation of the shape of a signal that indicates its characteristics (such as frequency and amplitude). A waveform is also called “waveshape” [5]. Timing diagrams consist of waveforms, timing specifications, and parameters, which define the shapes of individual signals and relationships among sets of signals.

Timing diagrams are widely used by electronic designers to describe the behavior of signals at interface circuits. The timing diagrams visualize timing information in a graphical format, which eases the realization of timing problems and relationships. They perfectly meet the requirements for a clear graphical specification of the I/O behavior of an interacting system [19, 20]. There are many powerful timing analysis tools that generate timing reports of valuable behavior information, but these reports are not easy to digest. Designers often convert the generated report information into timing diagrams (sometimes manually) to visualize the timing relations [8].

Figure 2-1 shows the intellectual property business model [11]. Encouraging the reusability of intellectual property (IP), the model connects the integrators (designers, engineers, etc.), with the providers (Manufacturers, vendors, etc.). IP providers can publish their IP interface-timing models described in timing diagrams [8]. IP integrators can browse the published IP models, examine the waveforms until they find an appropriate IP with desired specifications. Among a collection of potential IP candidates, the integrators can integrate the selected models into their design by simulating the timing diagram models using their own tools.

Figure 2-1: Systems-On-a-Chip Intellectual Property Business Model [11]

2-2 Timing Diagram Markup Language (TDML)

TDML was created as a standard representation of timing and waveform diagrams that describe component electronic characteristics and Intellectual Property (IP) behavior interfacing. The use of timing diagram as an interface specification to protect the Intellectual Property inherent to the design of component was a major motivation behind the creation of the TDML.

TDML was initially introduced under the management of the Silicon Integration Initiative (Si2). It was then developed by an Electronic Component Information

eXchange (ECIX) Working Group that included members from Chronology Corporation, Denali Software, IBM, Mentor Graphics/Interconnectix Business Unit, and SynaptiCAD [8]. The aim of the working group was to develop an interchange standard for timing diagram information. The timing diagram format is designed as a part of the ECIX electronic data sheet standard [8].

TDML is one of the markup language families, which are open industry standard languages based on the Extensible Markup Language (XML) [8]. TDML has the advantage of utilizing and accessing XML foundation services. XML compatibility enables interoperability support to achieve integration in a heterogeneous environment. Heterogeneity includes type, data, persistence, and execution models. As an XML extension, TDML can share information with the Pinnacles Component Information Standard (PCIS) [8]. PCIS is designed, as a business solution, to provide an efficient transfer and re-use of component information and System-on-Chip design, in both human and computer-sensible forms [8].

2-3 High Level Architecture (HLA)

“Under the authority of [DoD Directive 5000.59, “DoD Modeling and Simulation (M&S) Management”, January 4, 1994], and as prescribed by [DoD 5000.59-P, “DoD Modeling and Simulation Master Plan (MSMP)”, October 1995] I designate the High Level Architecture as the standard technical architecture for all DoD simulations”. - Dr. Paul Kaminski – (9/10/1996) [4]

The HLA establishes a common high-level architecture for simulation interoperation [17]. The HLA has been introduced to support modeling and simulation in a standardized

framework for Department of Defense (DoD) simulations. Although it was created to support military based simulations, the HLA is not a defense-oriented simulation framework. The HLA is a purpose independent simulation framework [18]. It facilitates the interoperability of all types of simulations and models [4]. The HLA can sustain simulations in a heterogeneous operating systems environment where the increasing demands of DIS (Distributed Interactive Simulation) and PADS (Parallel and Distributed Simulation) can be satisfied [27].

The HLA allows the composition of simulation systems from components. The components are called federates. The simulation which is composed of multiple federates is called a federation. The underlying operating system, which allows the interaction between federates, is called the runtime infrastructure (RTI) [4, 18]. The RTI offers services in many areas, including: federation management, declaration management, object management, and time management [4].

Federation management encompasses creating federations, joining federates into federations, resigning federates from federations, and destroying federations. Declaration management includes the publication and subscription of objects (events and data) shared among federates [18]. Each federate is responsible for identifying its publication and subscription interests to the RTI. Object management includes instance registration and instance updates by publishers and instance discovery and reflection by subscribers. Time management services to define time management policies and control time advances [4, 9, 10, 23].

The popularity of the HLA has been increasing rapidly. It has been approved as an IEEE Standard [2, 23], and NATO has adopted it as a standard framework for modeling and simulation [1, 26].

2-4 Simulation Reusability

The goal of simulation reusability is to reduce the time, cost and effort required to build a simulation. Achieving the goal relies on leveraging contributions, which have already been made; and thereby avoiding reinventing the wheel. As an example, an atmospheric simulation federate is a very essential component at several army based simulations and other civil based simulations [6]. Reusing an atmospheric simulation federate can reduce the time, cost and effort required to build such simulation. Another example is flight simulator models. They can often be reused from one version to another [24]. The reusability here needs verification to integrate the reused components in the new simulator model [24]. Another form of simulation reusability is the migration of a simulation from one platform to another. As an example, upgrading a non-HLA simulation into an HLA simulation to meet the DoD modeling and simulation purchase requirements.

A recent study attempted to determine the economic effect of the reusability on distributed simulations [6]. The study surveyed project managers in the DoD Modeling and Simulation Office who are responsible for HLA compliant simulations. Unfortunately many study questions could not be answered either for security reasons or because department policy does not track such information. The cost of the simulations ranged from \$150K to \$40 million. Maintenance costs ranged from \$10K and \$50K and

are included in the contract. The simulation lifespans ranged from 15 years to unlimited. The percent of reusability ranges from 50% to 100%. The study quoted the DoD HLA simulator business figure: “As of January 16, 2001, the US Army had 234 simulation models candidates for HLA application. They are either developed in inventory or being developed for military use.” The study concluded that reusability has a significant effect on the cost and worth of simulations. In the opinions of DoD modeling and simulation officials, the management and traceability of simulation reusability are not significantly identified under the economic basis of simulation contracts [6].

2-5 Related Studies

This section discusses three study cases that relate to the thesis research. The first case discusses the use of timing diagrams as testing tokens in the verification of a specific multiprocessor. The second case highlights the use of timing diagram specifications at hardware interfaces as an input for the generation of bus interface model. The third related study introduces an HLA framework tool for hardware designers. The study models system components using the DEVS formalism. This study deals with abstract interactions that occur above the level of details of signal interactions among hardware components.

Case 1: Hierarchical Random Simulation Approach for the Verification of S/390 CMOS Multiprocessors [25]

With the goal of obtaining a faster and better verification approach, IBM employed timing diagrams in a hierarchical verification study of memory control units, I/O adapters and processor interconnect units. The study introduced random executable timing

diagrams and associated CAD tool support. In this approach, the system level basic commands were stated (e.g. fetch data from memory, store data to memory) where a test case was written for each command. The test case included the input and output pattern of the command. The timing of the commands on the interfaces of each unit was specified in timing diagrams. A random selector decided when each test case started during the simulation where several of basic commands may be active at the same time. This might result in random complex traffic scenarios on each chip interface. The success of specifying commands' timing, on the interface of each unit, in timing diagrams motivates our study to specify systems behaviors in timing diagrams.

Case 2: The Automatic Generation of Bus-Interface Models [20]

Figure 2-2 depicts a study at Michigan University, which introduced a methodology to create VHDL Bus-Interface Models (BIM) automatically from timing diagrams and bus-state diagrams. The model generation time was reduced dramatically from months to days. An Intel iAPX80386 model was developed in less than one week. The methodology could be modified to produce models in a variety of Hardware Description Languages (HDLs). The generated HDL models of this study does not simulate above the HLA RTI platform. Our research could reuse the generated HDL models and simulate them above the HLA RTI.

Figure 2-2: Automatic Generation of Bus Interface Models by Using Timing Diagrams and State Diagram as Inputs [20]

Case 3: Computer system modeling at the Hardware Platform Level [22]

An approach based on DEVS and the HLA has been proposed to model computer hardware platforms. Each hardware component is modeled as a DEVS atomic formalism, and simulated under the HLA framework. The intent of the study is to provide a simulation tool for hardware designers that can be used during the product development phase. The study did not include the level of details of signal interaction scenarios between hardware components as a modeling factor. Our research could be used to extend these hardware models at the level of signal interactions scenarios.

Chapter 3: HLA Compliant System Simulator

This chapter is an overview of the thesis research. The chapter begins with a motivation section, which elaborates the benefits of enabling the generic creation of an HLA compliant system simulator in the hardware business market. The motivation clarifies and exposes some key thesis implications, including; the choice of timing diagrams to specify system behaviors, the choice of the HLA as a simulation platform, the overhead of approaching an object oriented modeling architecture, evolving existing models to include signaling scenarios, and finally the emphasis of simulation reusability.

The second section states the research questions that are relevant to the thesis. Some of the questions are answered in the subsequent chapters, and some remain for future research. The problem definition section defines the aspects of enabling an HLA compliant system simulator where system behaviors are specified in timing diagrams. In the fourth section, the thesis scope prescribes the limitations and boundaries of the thesis study. The fifth section identifies the research contributions of the thesis. Section six briefly introduces the solution. The solution assumes there are existing hardware models in which signaling scenarios have been abstracted into behavior atomic events. These atomic events are refined into sequences of signaling events. The solution includes a general architecture for an HLA compliant simulator, and demonstrates the simulator construction process in three consecutive modeling phases.

3-1 Motivation: a Tool for Modeling

The market for hardware components and Systems-on-Chip (SoC) Intellectual Property (IP) is rapidly increasing. Providers need to publish the behavior-interfacing models of their products [11]. These models describe the timing behaviors of hardware interfaces. Integrators need to try out components before agreeing to purchase. They do this by integrating the component model in their system, simulating the model and analyzing the timing results [8]. Introducing a tool, which enables the generic modeling of system behavior interfacing, is very beneficial in the hardware business cycle.

Significant effort has already been invested in modeling hardware systems. Some models ignore the details of signal interactions among system components while describing system behavior. The signal interaction scenarios are often abstracted into behavior atomic events [22]. Evolving the existing models by introducing signaling scenarios into system behaviors allows the reuse of the models to significantly reduce the cost of simulations.

Timing diagrams have a wide acceptance in specifying and analyzing the timing behaviors of hardware systems. Specialists find timing diagram specifications accurate and easy to digest [19]. Moreover, the existence of the Timing Diagram Markup Language enables the manipulation of timing diagrams in an interactive format. Allowing the use of timing diagrams to specify system behaviors in the modeling process is convenient for hardware components' providers and integrators.

The HLA supports the interoperability of simulations in heterogeneous distributed computing environments. It allows the reuse of simulations by composing a simulation

out of disjoint simulations [16, 18]. US army and similar defense organizations of other countries have mandated the HLA as their target simulation platform. As a result, there is a huge market demand for HLA based simulations (both military and civil simulations) [16]. Furthermore, the popularity of the HLA as a simulation platform is expected to increase. Defining an HLA compatible framework for simulations allows compatibility with the market demand and various organizations' mandates.

Due to the reduction of defense budgets around the world and economic slowdowns, budgets for simulations are shrinking. Therefore, reusability plays an important role when evaluating the cost efficiency of a simulation [6]. Furthermore, upgrading of simulations is a very common task due to expected changes in requirements over the lifespan of the simulations [13]. Many simulation contracts now include upgrading costs [6]. Introducing a reusable and scalable model would ease the upgrading process and allow the reuse of the model in further modeling business plans.

A large number of non-HLA simulations exist. Contracts, which mandate the HLA as a simulation platform, potentially exclude these simulations. Introducing a methodology for migrating a non-HLA simulation into an HLA simulation is worthy since existing simulations could be leveraged to meet contract requirements.

3-2 Research Questions

This thesis proposes a tool and methodology for generating an HLA-compatible framework for hardware system simulations. The target level of abstraction uses timing

diagrams to specify the behavior interfacing scenarios of the system components. The tool and methodology proposed in this thesis address the following questions:

- 1- How can an existing hardware simulation evolve to include the modeling of components' signaling scenarios?
- 2- How can a behavior-interfacing scenario be generated from a timing diagram specification?
- 3- How can the HLA simulation platform be employed to create a system simulator at the level of signal interactions?
- 4- How can a scalable, reusable form be established for a system simulation?
- 5- What might constitute a general methodology to upgrade a non-HLA simulation into an HLA compliant simulation?

3-3 Problem Definition

The main concern of this research is the modeling and simulation of hardware systems by specifying the interface signaling behaviors in timing diagrams and simulating the resulted models above the HLA platform. This thesis addresses the above research questions by focusing on the following problems:

There are existing models of hardware systems, which abstract the behavior interfacing scenarios of the system components into atomic behavior events. In these models, the atomic behavior events hide and ignore all signaling activities, and, signaling scenarios are discounted as a modeling factor [22]. The thesis intent is to evolve these existing models by introducing the signaling scenarios for the system components. Each model is

extended to include new multi-layer modules in which the atomic behaviors are refined into finer-grained sequences of signaling events. A major concern is the propagation of the finer-grained events through the new modules. The decomposition of atomic behaviors into their signaling events extends the existing models to simulate system behaviors involving component signal interaction scenarios.

Timing diagrams can describe and visualize timing relations, and design aspects of hardware system interfacing signals. The thesis intent is to model the system behavior interfacing. The modeling of a system behavior requires augmenting the behavior scenarios described in timing diagrams with user specifications.

The evolved model is intended to take advantage of being an HLA compatible model by following the HLA rules. The intent of the thesis is to provide an HLA-framework, which allows the propagation of signaling events across the HLA simulation platform. The signaling events must be transformed into HLA events for communications among federates.

Developing and updating models are expensive processes. One of the major factors in evaluating a simulation is its percentage of reusability and potential for further reuse [6]. The intent of the thesis is to introduce a framework and methodology that encourage both the reuse of existing models and the extension of models for future reuse. The choice of the HLA promotes reusability and extensibility since the HLA allows the composition of simulations [16, 18].

The HLA platform is mandated by significant key market customers [1, 2, 26]. The thesis intent is not only to provide HLA compliant system simulators, but also to introduce a methodology that allows non-HLA simulations to migrate onto HLA-based simulations.

3-4 Scope

The problems described above are too general to be solved in the scope of this research.

The following constraints are imposed to narrow the focus and direct the research.

Timing diagrams are central to the specification of interface behaviors. TDML format is used to represent timing diagrams in textual form. The free software provided at <http://www.timingtool.com/> was used to draw sample waveforms graphically, and to generate their associated TDML representations. The variety of AC-Characteristics associated with devices can be complex, and this research limits the breadth but supplies realistic examples by considering only AC-Characteristics that do not span more than one clock cycle.

Different HLA runtime environments can exhibit different behaviors. This research was carried out using the DMSO High Level Architecture (HLA) Run-Time Infrastructure RTI 1.3-Next Generation for Windows NT 4.0. Programming experiments and examples were developed using the Java programming language.

A case study example is used throughout the research to provide a realistic and practical context. The example is a simplified hardware system consisting of a processor component and a memory component. The research focuses on the read and write behaviors associated with the components.

Time and time management are fundamental to simulation interoperability. In this research, real-time simulation is not considered. Furthermore, the research is limited to event driven simulations that use the relative times associated with the signaling events

specified in timing diagrams. This approach ensures that the relative ordering among events is maintained; however, assigning absolute times to events is beyond the scope of this research.

The research lays the foundations for a tool to support the generation of simulation modules. Because of the exploratory nature of this research, only the requirements for the tool are addressed within the context of the demonstration case study. All examples have been generated manually. Where appropriate, suggestions have been made for the user interface of the tool. The further development and implementation of the tool is left for future work

The implementation of the proposed tool is an exploratory prototype. It is developed to explore the methodology presented in this dissertation. Therefore issues of validation and verification of the tool are beyond the scope of this work. Future research may consider a product version of the proposed tool and should address its formal validation and verification.

3-5 Research Contributions

- A methodology is proposed to model and simulate hardware systems. The methodology reuses simulations based on atomic interactions and refines them to include the signaling scenarios of the system behaviors. The signaling scenarios are specified in timing diagrams.

- An HLA-framework is defined to allow the HLA compatibility of the proposed simulation.
- The requirements for a support tool are proposed. The tool would automate the generation of the simulation modules needed to include signaling behaviors. The tool assumes timing diagrams are specified using TDML, and that the HLA is the target simulation environment.

3-6 Solution

Figure 3-1 shows the proposed architecture for an HLA compliant system simulator in which target component interactions can be simulated. Each target component is encapsulated in a federate along with a system model wrapper and an HLA Harness. The target component simulations are assumed to exist and are reused in the federate. The system model wrapper is where the modeling of signaling behavior and the integration of all signaling scenarios take place. The wrapper encapsulates all signaling behaviors using two interfaces: the Component Interface and the Communication Interface. The Component Interface regulates all events that take place between the system wrapper and the targeted component. From the target components' perspective, these events are atomic (e.g. read [address = 2000] or write [address = 2000, data =7]). The Communication Interface regulates all signaling events between the system wrapper and the HLA Harness. The signaling events are transformed by the HLA harness into HLA events that are dispatched as updates from publishing federate, and received as reflections at subscribing federates. The HLA Harness utilizes the HLA runtime infrastructure to communicate updates among component federates.

The remainder of this section overviews the solution by considering a general system modeling case. Initially, signal interactions are ignored and hidden behind system behavior atomic events. An HLA compliant system simulator is then a created specialized model, which accounts for signal interaction scenarios. The initial atomic events are refined into sequences of finer-grained signal interaction events.

The solution classifies the events into four types: behavior atomic events, signal interaction events, protocol events and HLA RTI events. The system wrapper accepts behavior atomic events and breaks them down into signal interaction events. At runtime, the sequences of signaling events are organized into a protocol that realizes an interactive scenario. The components involved in the scenario play different roles in the protocol. Protocol events are the system wrapper events exchanged by co-operating federates while progressing through a protocol. The protocol events are transformed into HLA RTI events for communication through the RTI.

The solution is illustrated using a case study of a simplified system of two components: a processor component and a memory component. The components interact using a “write” and “read” protocols.

Figure 3-1: HLA Compliant Simulator Architecture

3-6-1 Evolving Existing Models

The thesis introduces a modeling specialization by addressing signal interactions as a main concern in describing system behaviors. The approach assumes that the same system has already been modeled without concern for component signal interactions.

Evolving the existing models involves mapping the existing atomic events into protocols of signal interaction events, deriving the protocol events that must be exchanged, and mapping the protocol events to HLA RTI events.

The existing target component models ignore signal interaction scenarios, and interactions are accomplished using abstract atomic events. Figure 3-2 shows two target components, a processor component and a memory component, interacting using atomic read events.

Figure 3-2: Target Atomic Events of Existing Simulation

As shown in table 3-1, the write event is initiated by the processor component and processed by the memory component. The behavior initiation includes passing address and data parameters, which are required to accomplish the behavior. The memory component participates in the write event by storing the pair [address, data] in a table.

Processor	Memory
<pre>Void write (address, data) { Memory.write (address, data) }</pre>	
	<pre>Void write (address, data) { Memory Array[address] = data }</pre>

Table 3-1: Write Atomic Event

The example's read and write behaviors could be described in a message sequence chart as shown in figure 3-3, which shows the interactions as atomic events. Figure 3-3 includes no signaling interactions associated with the behaviors.

Figure 3-3: Messages Sequence Chart for Write/Read

Modeling the system by refining the behavior atomic events into protocols of signal interaction events is central to this thesis. The refined model enables studying the behavior and roles of components at the level of signal interaction protocols. Target

component interactions are realized by propagating events through the underlying system model wrappers, HLA harness and the RTI.

Figure 3-4 shows the events involved in an interactive scenario at the refined levels. Target components interact using the original behavior atomic events. An atomic event triggers the system model wrappers to perform the behavior-associated signaling protocol. In the first stage of translation, the atomic event is decomposed into signal interaction events (timing events). The signal interaction events are time ordered. In the second stage of event translation, the model realizes signal interaction events by generating protocol events. There is not necessarily a one-to-one correspondence between signaling events and protocol events

The system model wrappers realize the target component interaction via the protocol. The components are synchronized to execute the protocol in an ordered timing that depends on each component's role of involvement. The system model wrapper exchange protocol events via the underlying communication platform. In this research, the underlying platform is the HLA RTI. Protocol events are wrapped in HLA RTI events and communicated via the RTI. The model uses the HLA RTI as the target simulation platform, and federate communication occurs by wrapping protocol events into HLA RTI events.

Protocols progress through designed sequences of protocol events. Each protocol involves as atomic interaction among target components and therefore the protocol is triggered when one component issues the triggering event. Controlling federates publish protocol events as RTI update message The RTI reflects the published update to the

interested subscribing federates. The reflection is accomplished using RTI services and retranslating the communication events into protocol events. At some point during the protocol, the corresponding interaction must be carried out by invoking the other participating target component(s). Typically, an invocation of a participating component is triggered by the arrival of an associated protocol event.

Figure 3-4: Event Propagation

3-6-2 Write Scenario Example

Figure 3-5 shows the processor/memory write behavior as characterized by sequences of refined signal interaction events. Signaling events between the processor and memory components are represented by arrows. An arrow represents each signal interaction. The heads of arrows point to components, if that are observing the associated signals. The tails of arrows start from the component controlling the associated signal. Note that clock

signal is observed by both the processor and memory components, but the signal is controlled by neither.

The sample strip from figure 3-5 is enlarged in figure 3-6 to demonstrate the protocol events associated with signal interactions. Note that protocol events are represented by solid arrows, while signaling events are represented by dashed arrows. The sample starts at A by a falling clock signal event (CLK goes low). Both the processor and memory sense the event. The event is executed locally at each component due to the protocol specifications. The event sets a particular flag (specified by user) at each component, and both of the components respond by entering a new protocol state (state 3). Flags are used to monitor the local progress of protocols, and are discussed in more detail in chapter 4. At B, the signaling protocol requires the processor component to emit signals to define the data to be written to memory. The processor component checks the existence of the prerequisite runtime flags specified by the system protocol. Once the specified flags have been achieved, the processor component sends a protocol event at C indicating the processor has progressed in the protocol to the point where it has defined the data signal values (Update memory by D7-D0 data lines are Defined flag). Consequently, at D the processor component generates a protocol event wrapping the data value (Update memory buffer by Data value).

The signaling protocol requires the memory component to produce a “memory is ready” (RDY is high) signal event at E. An associated flag is generated, and the memory component acknowledges the processor at F by generating a protocol event (Update processor by RDY is High flag). At G, a falling clock signal event (CLK goes low).

Figure 3-6: Sample Example of Protocol Interactions

3-6-3 Methodology and Support Tool

On the design phase, the study approach tends to provide a model architecture, which allows the development of object-oriented simulations. The separation of concern principal is applied on the model architecture components. Loose coupling is intended among architecture components. The separation of concern principle is extended to include the propagation of simulation events.

The use of design patterns can yield a simpler, robust and reusable software architecture [29]. Three creational software patterns, Abstract Factory, Builder and Factory Method, are used in the design of the Protocol Model Factory. The Bridge structural pattern is applied to wrap the System Model by the Component Interface and the Communication Interface. The behavioral patterns, Observer pattern and State pattern, are used in the design of the System Model. The System Model instantiates the appropriate Command instance in the simulation runtime.

The simulation events are designed to have different types and format depending on the event origin and propagation direction in the model architecture. The design establishes rules of event translation, event wrapping and event decomposition.

Interoperability goals affect the design decisions. As a consequence the modeling approach involves the use of XML extensively in transmitting data among the architecture building blocks.

A tool is proposed to support the realization of the solution introduced above. The tool is structured into three successive building phases:

- Phase I : Building a protocol model
- Phase II : Building a system model wrapper
- Phase III : Building an HLA Harness

In phase I, each atomic interaction of the target components is broken down into a signal interaction protocol. A protocol model is constructed to encode the protocol-associated data. The protocol model distinguishes each component's role of involvement. The protocol model may also distinguish protocol behavior states, which the protocol passes through during its execution. Phase I data records are comprised of user design specifications, waveforms timing details and component information.

Phase II integrates all of the protocol models associated with targeted component interactions. The system model wrapper encapsulates the set of protocol models.

Phase III utilizes the HLA RTI simulation platform allowing component federates to communicate.

The highlights of each building phase are introduced below. Each building phase is discussed in detail in subsequent chapters.

3-6-3-1 Phase I: Building a Protocol Model

The goal of phase I is to build a protocol model that encodes a system behavior scenario described by timing diagrams. The information required to build a protocol model is organized as a set of segments. A segment is a group of related data records, which

contributes as a part of the protocol model. Segments are categorized based on their contained information (e.g. all waveforms events are contained in an associated segment of name: Events).

Figure 3-7 shows the proposed approach to construct a protocol model. Constructing the model takes two steps: data collection and protocol model initialization.

The data collection step is organized by segments. In a supporting tool, each segment would require a GUI application to enable editing, modifying, retrieving and storing the segment's inputs. The sources of segment information are target components specifications, user requirements and timing diagram details. Notably, phase I is the only phase, which requires user input.

Segment data records are stored in XML format to facilitate data interchange. Some input should be carried out in order, since some segment fields may rely upon other segments inputs (e.g. AC-Characteristic records depends on the signal events records).

In the protocol model generation step, each segment's data is integrated by the protocol model factory to produce a protocol model object.

The protocol model objects are input to phase II, where they are further integrated to produce a system model wrapper. The system model wrapper provides the structure and behavior necessary for the targeted components to interact using any of the protocol models.

Figure 3-7: Constructing a Protocol Model

3-6-3-2 Phase II: Building a System Model Wrapper

Recall from figure 3-1, that the system model wrapper hides the data and logic of all protocols behind the Component Interface and Communication Interface. The Component interface is the contract between the system model wrapper and any target component. Interactions that take place between the target component and the component interface are the behavior atomic events (e.g. read [18], write [18,1893]). The Communication Interface is the contract between any system wrapper and the

communication media harness where interactions are simply protocol events. Protocol events are specified and introduced in chapter 4.

The Component Interface is created by implementing invocation-compatible interfaces for the atomic events associated with the target components. The Communication Interface is created by enumerating the messages required to achieve protocol event interactions between the system model wrapper and the communication harness.

The system model factory processes all relevant protocol model objects to create a system model wrapper. The system model wrapper encapsulates the system model entities and mediates interactions among other architecture blocks. Phases I and II are independent of the communication media and the results of these phases could be reused, without modification, in a variety of runtime environments.

Phase III involves the construction of an HLA harness which works as a mediator between the system model wrapper and the HLA platform. The HLA harness interacts with the system model wrapper by exchanging protocol events. The harness further translates between protocol events and HLA RTI events.

3-6-3-3 Phase III: Building an HLA Harness

Figure 3-1 shows the HLA Harness as a mediator between the system wrapper Communication Interface and the HLA platform. HLA Harness provides (and hides) all the HLA RTI-related processing. It also provides a general event exchanging mechanism via HLA RTI, which allows the propagation of updates and reflections among federates.

Figure 3-8 shows the construction of a communication platform harness. Both the Communication Interface and the communication platform specifications are required to generate the harness. Communication platform specifications determine the nature of media events, the services provided by the platform and the strategies required to allow collaborative event progression. The Communication Interface contains the contracting calls, which should be implemented by the Harness.

Figure 3-8: The Construction of the Communication Media Harness

Figure 3-9 illustrates the mechanism of publishing and subscribing events between two federates in a federation. It shows the role of the HLA harness in the marshalling/de marshalling event propagation. Initially, the system model wrapper of the publisher federate issues a protocol event to the harness. The publisher federate HLA harness is responsible for conveying the protocol event through the HLA RTI platform by coding it

into an RTI update event. The updated event is reflected to subscribing federates. The subscribing federate's HLA harness is responsible for decoding the reflected RTI event back into a protocol event and delegating the reflected protocol event to the subscribing federate's system model wrapper.

Figure 3-9: Publisher/Subscriber Message Sequence Chart

Chapter 4: Phase I: Building a Protocol Model

The goal of phase I is to generate a protocol model that corresponds to an atomic event interaction among target components. A protocol model describes the atomic event in terms of a scenario of signal interactions among the components. Each participating component has a role of involvement, which is viewed as a sequence of relevant signal events. At each instant during the interaction, each component controls a set of signal events and observes a set of signal events. The resulting model is encoded in a protocol model file that can then be passed on to phase II.

Ideally, the development of protocol models would be supported by a tool. The tool would hide the details of building models by collecting the necessary data and then generating protocol models. The TDML files and the interfaces to target components do not provide sufficient information to generate protocol models automatically, and additional user input is needed to specify control flow and dataflow during signal interaction sequences.

This chapter presents the design of the general protocol model template that would be instantiated by the envisioned support tool. The design is described in terms of the input requirements to supply the tool with the needed data. The input is organized around logical segments that separate concerns during data collection. The segment-based approach enables extensibility and scalability of the protocol model development tool. Protocol entities and relations are constructed out of correlated data records not only on the segment level, but also across segments to finally generate the model. All segments

have associated data structures, which are instantiated collectively at runtime to create the protocol model.

The construction of a protocol model consists of two steps: the data collection step and the generation step, as elaborated in figure 4-1. The data collection step is organized into segments that each deals with a collection of related data. The segments are discussed in details later in this section. The data collected for each segment is saved in a file in XML format. The Protocol Pointer Table records the names and locations of all of the segment files associated with a particular protocol. In the generation phase, the Protocol Modeling Factory reads all of the segment XML files and generates the protocol model. The protocol model is saved as a file for later use in phase II.

Figure 4-1: Protocol Model Construction Process

Figure 4-2 describes the data collection step that would be realized by the support tool. The sources of data inputs are: system signal interactions (provided in timing diagrams), user input and components' information. The inputs are processed by GUI applications. The GUI applications allow the user to input/edit/modify/save and retrieve data. The GUI applications use the XML-based encoder to store input information in an XML-based format, with a unique file for each segment. Each segment file is also recorded in the Protocols Pointer Table.

Figure 4-2: Data Collection Step

Figure 4-3 describes the protocol model generation step for the support tool. The protocol model factory builds a protocol model out of the segment data collected in the previous step. The protocol model factory has data structures corresponding to each segment, and an XML-decoder. The Protocols Pointer Table is used to access the XML-based segment files. The XML decoder is used to extract data from the segment files. The extracted data is then used to instantiate the data structures. The populated data structures are then processed to generate the protocol model.

The XML-based encoder and decoder in figure 4-2 and 4-3 use a Document Object Model (DOM) model to define the XML details. The DOM API(s) can be used to construct, query, traverse and manipulate XML documents programmatically.

Figure 4-3: Protocol Model Generation Step

The tool would support the construction of a Protocol Pointer Table at two levels: individual protocol models and collection of protocol models. At the individual protocol model level, a link to each segment XML file is saved in a segment record of the form [segment name, file path] (eg. [AC-Characteristics, Root/characteristics1.xml]) The segment records that constitute a protocol are saved in a segments table, which is in turn saved in a protocol record of the form [protocol name, segments table]. The protocol Pointer Table contains a set of protocol records, and will be reused later in phase II.

The remainder of this chapter presents the individual segments. Each segment is discussed in terms of: the role of segment's information in the protocol model, the main entities associated with the segment, a proposed representation layer (GUI), the XML format of the associated segment file, and the resulting data structure used in the Protocol Model Factory.

In some cases, the discussion includes a brief overview of how the segment information would be utilized at simulation runtime (vs. tool runtime). This is necessary to understand the role of the information in the protocol model.

4-1 Events Segment

The events segment maintains records of system waveform information. Signal interaction events are the main concern of this segment. Components integrate by exchanging signal interactions according to a specific set of rules that comprise protocol's behavior. Figure 4-5 represents a simplified behavior read for a system example of a processor component and a memory component. The read protocol is described in timing diagrams by a set of signals.

4-1-1 Data Collection

Figure 4-4 summarizes the data collection process of Events segment. The user inputs waveforms using a GUI (timing diagram editor). The GUI can save the waveform information in a TDML file format (XML-based format).

The next paragraphs demonstrate the used GUI as a data visualization tool and how to represent events segment at data collection process.

Figure 4-4: Events Segment Data Collection Step

Events segment visualization can depend on available timing diagram tools, which provide waveform editors. These tools have predefined elements to transfer waveforms into a descriptive file format. For the implementation example, a free TDML-based timing diagram tool is used, which is located at <http://www.timingtool.com/>. Figure 4-6 shows the timing diagram tool editor screen after the read protocol in figure 4-5 has been to input. The tool can translate all represented waveforms into a corresponding TDML file format. The generated TDML file keeps track of the number of waveforms, clock signal details and each waveform timing details

Figure 4-5: Simplified Read Behavior Waveforms

Figure 4-6: Timing Tool Simplified Read Behavior Visualization

4-1-2 Segment Data Structures

As shown in figure 4-7, the Protocol Model Factory assigns the Diagram package as corresponding data structures for the Events segment. Each change of state in a TDML recorded waveform is saved in an Edge data structure. All Edges in a waveform are saved in a Signal data structure. The Edges are sorted by their relative time of occurrence. The factory processing results in a set of Signals that collectively define ordered sequences of Edges.

Figure 4-7: Events Segment Corresponding Data Structures

Figure 4-8 shows the UML class diagram of the Diagram package data structure. The Diagram module represents all provided waveforms in the input TDML file. It has a collection of Signal instances representing waveforms. A Signal has a collection of Edge instances representing the individual events in a waveform. Each Edge is identified by its signal name, a unique ID, its signal status and its time of occurrence (e.g. signal is Select, ID = sel18, status = High, time = 600). The Transform module represents the XML-decoder. It extracts data from XML based format TDML file using Diagram data structure to provide two important services: the clock signal name and a time ordered collection of all of the protocol events.

Figure 4-8: Diagram Package Structure

4-2 Component-Signal Relation Segment

The Component-Signal Relation segment keeps track of the relation between a component and a signal in a protocol. The relation takes one of the forms:

- A component controls the signal
- A component senses the signal
- A component neither controls nor senses the signal

Identifying this relation between a component and a signal is essential in extracting the component's role of involvement in a protocol. Since each event is associated with a signal, and each signal is controlled by a component, then extracting any component's role of involvement represented in a sequence of events is possible.

4-2-1 Data Collection

The study proposes a GUI application to collect this segment's data records and save them into an XML file. Figure 4-9 introduces an illustrative example of a Component-

Signal Relation presentation layer. In the example, a Range Window is used to limit the parts of the signals that are controlled or sensed (observed) by each component. The visualization includes the component involved, in this case the processor and the memory components, and the waveforms obtained from the Events segment. The user can draw a Range Window over a signal to indicate a set of events that will have a particular status in the protocol. The Range Window is then linked to the components to specify the relation. For example, the CLK signal is observed (sensed) by both components. This is indicated by the dotted arrows connecting the components to the Range Window over the CLK signal waveform. The S signal is controlled by the processor as indicated by the solid arrow between the processor and the Range Window over the S waveform. Note that the signals have a time base, however the lines extending from the components do not (despite their similarity to message sequence chart notation). This layer could be built upon the Events segment presentation layer to display time as a possible factor in the relation.

Figure 4-9: Presentation Layer of Component-Signal Relation

Figure 4-10 shows a sample of a Component-Signal Relation XML file for the read protocol (figure 4-5). The sample does not show the entire file but just some elements to demonstrate the main structure.

The file begins with the root element `<protocol_relations>` that has three sub-elements:

- `<Protocol_Name>`: stores the protocol name.
- `<Protocol_ID>`: stores the unique ID of the protocol
- `<signals_states>`: stores all protocol relations in its relation sub-elements

The relation element structure is as follows:

<component_Name>	: Stores a system component name
<signal_name>	: Stores protocol signal name
<control>	: Accepts either 1 or 0 to indicate that the system component controls this signal or not.
<sensor>	: Accepts either 1 or 0 to indicate if the component senses the signal or not.

```

<?xml version="1.0" ?>
<Protocol_relations>
  <Protocol_Name>read</Protocol_Name>
  <Protocol_ID>8001</Protocol_ID>
  <signals_states>
    <relation>
      <component_Name>Processor</component_Name>
      <signal_name>Select</signal_name>
      <control>1</control>
      <sensor>0</sensor>
    </relation>
    <relation>
      <component_Name>Processor</component_Name>
      <signal_name>RDY</signal_name>
      <control>0</control>
      <sensor>1</sensor>
    </relation>
    <relation>
      <component_Name>Processor</component_Name>
      <signal_name>CLK</signal_name>
      <control>0</control>
      <sensor>1</sensor>
    </relation>
    <relation>
      <component_Name>Memory</component_Name>
      <signal_name>Select</signal_name>
      <control>0</control>
      <sensor>1</sensor>
    </relation>
    <relation>
      <component_Name>Memory</component_Name>
      <signal_name>RDY</signal_name>
      <control>1</control>
      <sensor>0</sensor>
    </relation>
    <relation>
      <component_Name>Memory</component_Name>
      <signal_name>CLK</signal_name>
      <control>0</control>
      <sensor>1</sensor>
    </relation>
  </signals_states>
</Protocol_relations>

```

Figure 4-10: Component-Signal Relation XML Structure

This research does not consider a case where a component might alternate between observing and controlling a signal during a protocol. Future work should extend this

segment to allow the signal and the relation to be partitioned into regions. The extension would require one additional sub-element of the relation: <From>, which defines the start time of a region and the relation status for the region. The start time could be represented by a relative number of timing units, or could be represented by a signal event.

4-2-2 Segment Data Structure

For this segment, an associated package is implemented to enable translating segment data records into Component-Signal Relation instances. A Component-Signal Relation object stores information about a component/signal relation and provides access services for: component name, signal name, and relation status (i.e. control, observe, neither).

4-3 AC-Characteristics Segment

The AC-Characteristics of waveforms define how instances of the waveforms may differ from those shown in timing diagrams. The tolerances on setup and hold times are examples. AC-Characteristics are expressed in terms of timing intervals. A signal constrained by an AC-Characteristic may satisfy any timing behavior allowed by the characteristic.

To simplify the discussion of timing relationships like AC-Characteristics, protocols are often partitioned into states. State transitions are triggered by signaling events, and the transitions provide convenient anchor (reference) point for specifying relative timing constraints. Protocol state information is recorded in the Protocol State segment, and is discussed later in section 4-5.

4-3-1 Data Collection

Specifying an AC-Characteristic requires user input to: identify protocol states, name the characteristic, identify a trigger event that anchors the relevant time interval, identify the target event that is constrained relative to the anchor, and specifying the minimum and/or maximum duration of the interval.

The AC-Characteristics segment's GUI application would allow AC-Characteristics to be graphically represented in a layer above the waveforms layers. The presentation layer allows the protocol to be divided into states, and each state is automatically given a unique ID.

Figure 4-11 shows an AC-Characteristics segment for the read protocol example. The protocol has been partitioned into states. AC-Characteristics have been added using a label and pairs of arrows to indicate the relevant interval. Consider the RD/WR signal. The first AC-Characteristic for the signal is labeled TRS, and occurs in State 1. The relevant interval begins with the rising edge of the signal and ends with the first subsequent falling edge of the CLK signal. Although the duration data for the characteristic is not shown, suppose that the minimum duration is 40 time units, and the maximum duration has not been specified.

Figure 4-11: AC-Characteristic Presentation Layer

After inputs are obtained, the AC-Characteristics information is stored into an XML file.

Figure 4-12 shows a portion of the AC-characteristic segment XML file (it does not show the entire file). The example shows the XML structure for the TRS characteristic described above. Line 1 shows the root element of the XML document, which comprises all document elements. Line 2 has an element to identify the protocol. Line 4 shows

<All_Characteristics> element, which contains all the AC-characteristic elements of the protocol. From Line 5 to line 15, the figure demonstrates <characteristic> element, which describes a single AC-Characteristic.

Line	
	<?xml version="1.0" ?>
1	<Protocol_Characteristics>
2	<Protocol_Name>read</Protocol_Name>
3	<Protocol_ID>8001</Protocol_ID>
4	<All_Characteristics>
5	<characteristic>
6	<characteristic_name>TRS</characteristic_name>
7	<characteristic_Description>Read Setup Time</characteristic_Description>
8	<signal_name>RDnotWR</signal_name>
9	<component_name>Processor</component_name>
10	<MinValue>40</MinValue>
11	<MaxValue>-1</MaxValue>
12	<sourceEvent>RDnotWR,1</sourceEvent>
13	<targetEvent>CLK,0</targetEvent>
14	<uniqueID>1004</uniqueID>
15	<StateID>7001</StateID>
16	</characteristic>
17	</All_Characteristics>
18	</Protocol_Characteristics>

Figure 4-12: AC-Characteristics XML Sample

The sub-elements of <characteristic> maintain important AC-Characteristics attributes. At line 6 <characteristic_name> element carries out the name attribute while <characteristic_Description> at line 7 describes the AC-Characteristic. <signal_name> at line 8 stores the associated signal name while <component_name> element at line 9 tracks the hosting component. Line 10 has an element that holds the minimum time required for the AC-Characteristic while line 11 shows <MaxValue> element, which restricts the time by an upper limit value. The example value is (-1), which means that no limit was specified. <sourceEvent> element at line 12 records the triggering signal event

of the AC-Characteristic while <targetEvent> at line 13 tracks the end signal event of the AC-Characteristic. <StateID> element at line 15, points at the protocol state at which the AC-Characteristic begins. <uniqueID> element at line 14 stores the unique ID of the AC-Characteristic which is assigned automatically by the presentation layer.

The sample example from line 12 to line 15 means that the AC-Characteristic is expected in the protocol state ID = 7001 (protocol state 1). The AC-Characteristic begins when the **RdnotWR signal** becomes **high** and ends when the **falling clock** edge takes place.

4-3-2 Segment Data Structure

At the protocol model generation step, the protocol model factory extracts the AC-Characteristic information out of the segment XML file and instantiates AC-Characteristic package. The package contains the required structures to maintain the necessary data to simulate AC-Characteristic behavior at runtime. Figure 4-13 shows the UML structure of the AC-Characteristic package. An AC-Characteristic instance refers to both the protocol and the protocol state(s) where it occurs, and tracks the runtime signal events (Edges), which mark its start and end.

Figure 4-13: AC-Characteristic Package in an UML Diagram

4-4 Flags Segment

A major concern in a simulation of signaling interaction is monitoring the progress through a protocol such that each component generates signals that satisfy the timing diagram specifications. The flags segment introduces flags to represent conditions that indicate incremental progress through a protocol. In this research, the source of a flag condition is either the occurrence of a signal event, or the satisfaction of an AC-Characteristic. A flag may also have prerequisite flags that must also be satisfied to meet the flag's condition. The value of a flag is either TRUE, indicating that the condition has occurred, or FALSE, indicating that the condition has not yet occurred.

Flags are abstract conditions that are used at runtime to control progress through a protocol, and to ensure that each component's behavior satisfies the orderings specified in the timing diagrams. To describe more complex conditions, a flag may have additional flags specified as prerequisite conditions. The use of flags effectively elevates the implementation of protocol interactions from the level of signal interactions to a level of flag exchanges. Instead of exchanging signal edges, federates exchange flags that have been assigned to specific edges in the protocol. This allows easier interpretation of received (flag) interactions in terms of progress through a protocol.

Flags are the logical markers that synchronize the progress of all components through a protocol. To appreciate the role of flags at runtime, it is important to remember that each federate participating in a protocol has information about all of the flags that are relevant to the protocol. In this research, a flag can have two forms: a signal edge (recall Events segment), or an AC Characteristic (recall AC-Characteristics segment). Both forms of flags have trigger edges. For the edge form, the trigger edge is the specified edge. For the AC-Characteristic form, the trigger edge is the edge that defines the end of its interval. When a component responsible for generating a trigger edge determines that all conditions associated with a flag are met, then it sends the flag's unique identifier to any components waiting for the flag. A flag is said to "occur" when it is exchanged. Sending and receiving components record (or "mark") information about relevant flags as they occur, and thereby advance through a protocol in unison.

4-4-1 Data Collection

The flags segment relies on user input to define flags and prerequisite condition. Flags presentation layer should allow users to input all required fields to identify a flag. Figure 4-14 is an example of the proposed flag presentation layer. It depends on the graphical existence of the protocol waveforms, AC-Characteristics and the protocol states. A flag is presented by framing its source. Arrows are drawn from any prerequisite to a flag.

Figure 4-14:Flags Presentation Layer

Figure 4-14 shows some flag examples for the read protocol on the Flags presentation layer. The presentation layer generates the sources of all potential flags automatically, and several (user drawn) prerequisite arrows are shown. Two flags will be discussed in more detail: the first S signal flag, and the first D7-D0 signal flag. The S signal flag is an

AC-Characteristic flag associated with the TSS AC-Characteristic. The rising S signal begins the characteristic interval, and the falling CLK edge is the flag trigger edge. There are three prerequisite flags for the D7-D0 (data lines are verified) flag: the TSS AC-Characteristic, the RD/WR signal rising edge, and the A15-A0 (address) lines being asserted.

The presentation layer saves flags in an XML file. Figure 4-15 demonstrates the proposed XML structure of the example flags discussed in figure 4-14. Line 2 contains the root element of the entire XML structure <Protocol_Flags>. The root element has three direct sub-elements: <Protocol_Name> that identifies the protocol of concern, <Protocol_ID>, <all_flags>: that comprises all segment flags.

From line 6 to line 17, is the description of a flag element. This flag has an ID = 4007 (line 7). The tool presentation layer automatically generates the flag ID. The flag source is of type signal event (line 9) and the source is “D7-D0 is verified” signal event (line 8) which take place at protocol state of ID = 7003 (line 11). This flag has a collection of prerequisite preceding flags. Their IDs are [4017, 4003 and 4004] (lines 13, 14 and 15)

```

line
1    <?xml version="1.0" ?>
2    <Protocol_Flags>
3        <Protocol_Name>read</ Protocol_Name>
4        <Protocol_ID>8001</ Protocol_ID>
5        <all_flags>
6            <flag>
7                <flag_ID>4007</flag_ID>
8                <flag_owner>D7-D0,V</flag_owner>
9                <flag_type>1</flag_type>
10               <flag_component_name>Memory</flag_component_name>
11               <flag_state_ID>7003</flag_state_ID>
12               <causes>
13                   <flag_cause_ID>4017</flag_cause_ID>
14                   <flag_cause_ID>4003</flag_cause_ID>
15                   <flag_cause_ID>4004</flag_cause_ID>
16               </causes>
17           </flag>
18       </all_flags>
19   </Protocol_Flags>

```

Figure 4-15: Sample from Flags Segment XML File of a Read Protocol Example

4-4-2 Segment Data Structure

Figure 4-16 shows the Flag package in a UML Diagram. The package has aggregations with AC-characteristic, Edge, Protocol State and itself. The protocol model factory decodes the flags segment data records out of its XML file and instantiates the protocol flags at runtime. The flags' fields are updated during the runtime simulation of the protocol. The updated flags become the triggering aspects of the protocol signal interaction consequences (e.g. beginning of dataflow, declaring a new protocol state, invoking an atomic original event,...)

Figure 4-16: Flag Package Structure

4-5 Protocol States Segment

As described earlier, the protocol execution period can be partitioned into protocol states. Since studying the protocol in its different phases of execution may be an issue of concern, it is important to assign a segment to preserve protocol states records. For each protocol state, the protocol requires a specific set of events and associated AC-Characteristics to take place. Consequently, a set of flags can be used to specify and track the relevant conditions at runtime. When the end event (flag) for a state occurs, the protocol moves to another state depending on the flags achieved during the state. At the end of last protocol state, the protocol has completed all required signaling.

4-5-1 Data Collection

For each protocol state, the user must specify all expected edge events, AC-Characteristics, protocol states attributes (e.g. wait states may occur), and different probable next states depending on the protocol runtime scenarios. The existence of previous segments data records (AC-Characteristics, Flags and Events) would ease the matter since the user can refer to fields defined previously, instead of defining them again. The protocol state presentation layer may be built as a simple extension to the flags presentation layer. The extension allows the association of a flag with the end of protocol state, and the list of prerequisite flags for each state.

The XML structure that holds a protocol state is heavily packed. For that reason, the protocol state XML structure is introduced in two figures: figure 4-16 which demonstrates the main XML structure and figure 4-17 which discusses the responsible elements for determining the next protocol state.

The sample example shown in figure 4-16 stores the fields required to represent a protocol state. The state gets an automated ID of 7001 (line 9). The state is described as the first protocol state where setup(s) are expected (line 10). This protocol state does not expect a wait state (line 11). At the beginning of the state, the processor component is the master component while the memory component is the slave component (lines from 12 to 19). The expected signal edges in this protocol state are: RdnotWR signal becomes high (line 25), Address lines signal is verified (line 26) and a falling clock edge (line 27). The expected AC-Characteristics at this protocol state are: read setup time and Address setup time (lines 20 to 23).

```

line
1      <?xml version="1.0" ?>
2      <Protocol_States>
3          <Protocol_Name>read</ Protocol_Name>
4          <Protocol_ID>8001</ Protocol_ID>
5          <all_States>
6              + <Protocol_State>
7              + <Protocol_State>
8              <Protocol_State>
9                  <State_ID>7001</ State_ID >
10                 <state_Description>
11                     Protocol State 1 where setup(s) are expected
12                 </state_Description>
13                 <Expected_Wait_Cycle>0</ Expected_Wait_Cycle >
14                 <LeadStatus>
15                     <component_Name>Processor</component_Name >
16                     <Master_Case>1</Master_Case>
17                 </LeadStatus>
18                 <LeadStatus>
19                     <component_Name>Memory</component_Name >
20                     <Master_Case>0</Master_Case>
21                 </LeadStatus>
22                 <associated_Characteristics>
23                     <characteristic>TRS,1012</characteristic>
24                     <characteristic>TAS,1006</characteristic>
25                 </associated_Characteristics>
26                 <expected_Events>
27                     <event>RDnotWR,1</event>
28                     <event>Address,V</event>
29                     <event>Clk,0</event>
30                 </expected_Events>
31             + <next_state_dictionary>
32         </Protocol_State>
33     + <Protocol_State>
34     + <Protocol_State>
35         //possible more protocol states
36     </all_States>
37 </Protocol_States>

```

**Figure 4-17:Protocol States XML Main Structure
Sample from Read Example**

Figure 4-18 shows the element of tag name “next_state_dictionary” which holds the potential cases that may occur as a next protocol state. Each case is represented by an element of tag name “case”. The “case” element has a collection of prerequisite flags that form the condition for advancing to a particular next state. The flags achieved at runtime are accumulated and compared with the case prerequisites to decide which case specifies the next state. The <next_StateID> field of the case element identifies the associated next protocol state. One of the cases could be a termination case.

Figure 4-18 shows a <case> element, which is a sub-element of <next_state_dictionary>. The element is taken from the read example for a protocol state of ID 7003, which might expect a wait state. The example should be interpreted as:

```
IF {
    TDS Data setup Time AC-Characteristic flag occurred (line 5)
        AND
    TRDYS RDY setup Time AC-Characteristic flag is occurred (line 9)
        AND
    Data,V Data lines signal is verified flag occurred (line 13)
        AND
    RDY,1 RDY signal is high flag occurred (line 17)
        AND
    Clk,0 Clock falling edge flag occurred (line 21)
}
THEN
    Next protocol state is of protocol ID 7004 (line24)
ELSE
    Check other cases
```

line	
1	+ <next_state_dictionary>
2	<case>
3	<pre_Condition_Flags>
4	<Condition>
5	<flag>0,4034,TDS,1014,1</flag>
6	<effect>AND</effect>
7	</Condition>
8	<Condition>
9	<flag>0,4020,TRDYS,1008,1</flag>
10	<effect>AND</effect>
11	</Condition>
12	<Condition>
13	<flag>1,Data,V,4030,1</flag>
14	<effect>AND</effect>
15	</Condition>
16	<Condition>
17	<flag>1,RDY,1,4008,1</flag>
18	<effect>AND</effect>
19	</Condition>
20	<Condition>
21	<flag>1,CIK,0,4009,1</flag>
22	<effect>AND</effect>
23	</Condition>
24	</pre_Condition_Flags>
24	<next_StateID>7004
26	<Priority>1
27	</case>
28	+ <case>
29	+ <case>
30	+ <case>
31	+ <next_state_dictionary>

**Figure 4-18: XML Structure for Next Protocol State
Sample from Read Example**

4-5-2 Segment Data Structure

Figure 4-19 shows a UML diagram of the proposed protocol state package, which demonstrates some of the package's associations. Other package details are demonstrated later in figure 4-20.

As shown in figure 4-19, the protocol state package can hold a collection of expected Edges (signal events), a collection of expected AC-Characteristics and a collection of expected Flags. The protocol state package also has a pointer to its hosting protocol.

Figure 4-19: Protocol State Package Part A

Figure 4-20 demonstrates other entities of the proposed protocol state package. The protocol state keeps a pointer to another protocol state (its next state). For each scenario, there is an instance of NextStateCase which points to a specific next protocol state. The NextStateCase mandates an expected collection of runtime flags as StateConditions.

Figure 4-20: Protocol State Package Part B

4-6 Dataflow Segment

The dataflow segment establishes the linkages between target component interactions and protocol signaling. The segment identifies the flow of target component data through participating system model wrappers, and links the dataflow to the invocation and termination of target component interactions.

In this discussion, “source” will be applied to a federate that generates data (e.g. source target component, source federate, etc.), and “destination” will be applied to the federate that receives the data. A federate may be both a source and a destination during a protocol. The read protocol will be used as an example throughout the dataflow discussion. In the read protocol, the processor initiates the protocol and supplies a memory address. The memory receives the address, and generates the memory contents stored at the address. The contents must in turn be communicated and received by the processor. The processor is the source of the address, and the destination of the contents.

Similarly, the memory is the destination for the address, and the source of the contents. The dataflow segment must account for target component interactions as the sources and destinations of dataflow, the internal buffering of data in the system model wrappers, and the integration and extraction of data with protocol signaling events.

Since the target components view a protocol as a single action, all of the component-level data exchanged in a protocol can be expressed using a single function-like signature. The function's name represents the protocol name. The name is followed by a bracketed argument list that represents the data that must be supplied by the federate that initiates the protocol. The function's return value represents any data that is returned by other federates participating in the protocol. Figure 4-21 shows the signature for the read protocol. The single bracketed argument represents the address value supplied by the processor. The return value represents the returned memory contents. The signature identifies all target component data exchanged during a protocol, and distinguishes between the data that is supplied by the component that initiates the protocol versus the data that is supplied by other participating components.

Figure 4-21: Read Protocol Signature

The target components interact with their local system model wrappers to receive and supply data. Data supplied by a source target component must be buffered locally in the system wrapper until the appropriate point in the protocol when the data can be communicated to the destination federate. When data reaches a destination federate, it must again be buffered in the destination system model wrapper until an appropriate point in the protocol for it to be delivered to the destination target component. A system model wrapper must contain a buffer variable for each data item for which the associated target component it is either a source or a destination.

The function-like signature used to describe the component-level data exchange during a protocol also defines the variables required by system model wrappers to buffer component data during the protocol. For the read protocol, the processor federate's system model wrapper must have variables to buffer generated address data and received memory contents data. Similarly, the memory federate's system model wrapper must have variables to buffer received address values and generated memory contents.

The communication of component data between federates is accomplished using the signaling protocol. The read protocol uses the A0-A15 signals to communicate address values, and the D0-D7 signals to communicate memory values. On the source federate side, each data value must be integrated into the protocol. This involves assigning the correct values to the appropriate signals at the appropriate point in the protocol sequence. The buffer variables (introduced above) hold the data until the appropriate point in the protocol. On the destination side, the data must be extracted from the signals, and buffered until the appropriate point in the protocol to pass the values to the destination target component.

From the dataflow perspective of participating system model wrappers, each value navigates a path through the source and destination wrappers in which there are three significant events:

- 1- Source: This event occurs in the source wrapper when the data arrives from the target component and must be internally buffered.
- 2- Communication: This event corresponds to the point in the protocol at which the data value is communicated between federates. This event involves actions in both the source and destination wrappers. The source wrapper must send the value. The destination wrapper must receive the value and buffer it internally.
- 3- Destination: This event corresponds to the point in the protocol at which the target component reads the data value. At this point, the buffered data value must be passed from the destination wrapper to the destination target component.

Figure 4-22 shows the dataflow of a protocol variable between a source component (Component 1) and a destination component (Component 2). The Variable path consists of a source event, a communication event, and a destination event. The initiation of a protocol by a target component is a source event at which data is supplied according to the argument list in the protocol signature. The subsequent communication and destination events correspond to signals defined in the protocol. The destination wrapper activity triggered by the destination event includes interacting with the destination target component to deliver the data and complete the data path. The destination wrapper views this as an atomic target component interaction. If the protocol signature includes a return value, then the destination target component must generate the return value during this

interaction and pass it back to the wrapper. In this case, a source event has occurred and the associated return value must be communicated using protocol signaling. The destination event associated with the return value path corresponds to the completion of the target component interaction that originally initiated the protocol.

Figure 4-22: Protocol Variable Data Path

The dataflow segment must also deal with cases where the completion of a target component interaction does not involve a return value. In the discussion above, the target component interaction that initiates the protocol does not complete until the destination event associated with the return value. The dataflow segment must include the

identification of a “termination” signaling event corresponding to the point at which the target component interaction terminates.

4-6-1 Data Collection

As shown in figure 4-23, Dataflow presentation layer should allow users to identify protocols signatures including the signature variables, the data paths of the signature inputs and outputs, and the data events along each path. The inputs are managed by a visualization editor to store the segment information at three XML files: Signature, Data Events and Paths.

Figure 4-23: Data Collection of Dataflow Segment

Figure 4-24 shows the structure of read <signature> XML element. Line 2 states the protocol name. Line 4 contains the path of the read interface file, which describes the read signature. The list of read input parameters is described in the lines from 5 to 11. The list has one variable of type integer, its ID is 6003, and its index, in the input list, is 0. The read output information is described in the lines from 12 to 18. It is an integer, its ID is 6004, and its index in the output list is 0.

```

1    <Signature>
2        <command_name>read</command_name>
3        <Protocol_ID>8001</Protocol_ID>
4        <interface_file_path>ReadInterface.java</interface_file_path>
5        <input_list>
6            <variable>
7                <variable_index>0</variable_index>
8                <variable_id>6003</variable_id>
9                <variable_type>int</variable_type>
10           </variable>
11        </input_list>
12        <output_list>
13            <variable>
14                <variable_index>0</variable_index>
15                <variable_id>6004</variable_id>
16                <variable_type>int</variable_type>
17            </variable>
18        </output_list>
19    </Signature>

```

Figure 4-24: Read Signature XML Structure

Figure 4-25 shows a <Data_Event> XML element structure, which stores the information of a read signature data event. It is a source data event and its triggering flag ID is 4000. The data event source is the Processor target component and the destination is the Processor system model wrapper.

```

1    <Data_Event>
2        <Data_Event_Type>Source</Data_Event_Type>
3        <Trigger_Flag_ID>4000</Trigger_Flag_ID>
4        <Data_Event_ID>5001</Data_Event_ID>
5        <From>
6            <Name>Processor</Name>
7            <Status>1</Status>
8        </From>
9        <To>
10           <Name>Processor</Name>
11           <Status>0</Status>
12       </To>
13   </Data_Event>

```

Figure 4-25: <Data_Event> XML Element

Figure 4-26 shows the address <Signature_Variable> XML element structure in a read protocol signature. The element stores information about the “address” signature variable, its initial hosting component and its associated data path. The variable ID is 6003, its name is address, and its initial target component is the Processor. The address associated data path has a sequence of ordered data events. The first data event ID is 5001 and its protocol global order is 1. The second data event ID is 5002 and its global order is 2. The third data event ID is 5003 and its global order is 3.


```

1      <Signature_Variable>
2          <Variable_ID>6003</Variable_ID>
3          <Variable_name>Address</Variable_name>
4          <description>
5              an integer in the range of [0,2^15] that points to
6              a specific address in memory
7          </description>
8          <hostComponent>Processor</hostComponent >
9          <Variable_Path>
10             <Flow>
11                 <Event_Order>1</Event_Order>
12                 <Data_Event_ID>5001</Data_Event_ID>
13             </Flow>
14             <Flow>
15                 <Event_Order>2</Event_Order>
16                 <Data_Event_ID>5002</Data_Event_ID>
17             </Flow>
18             <Flow>
19                 <Event_Order>3</Event_Order>
20                 <Data_Event_ID>5003</Data_Event_ID>
21             </Flow>
22         </Variable_Path>
23     </Signature_Variable>

```

Figure 4-26: Address Path XML Structure in Read

As previously discussed in the beginning this section, Dataflow Presentation Layer allows users to define the signature of a protocol and the dataflow of its associated variables. The following paragraphs in this section demonstrate the working of the dataflow presentation layer, in detail, with the help of a concrete example. Figure 4-27 depicts the read protocol signature (figure 4-21), and the dataflow of its associated variables. Defining the behavior of signature variables depends upon the waveforms layer, since it is the main source of triggers for dataflow. The dataflow presentation layer shows the behavior of signature variable with a set of symbols listed in Figure 4-27. The

read protocol has two signature variables; address and contents. The lifecycle of the signature variables is summarized as follows.

- The read protocol is invoked by the processor target component and a data value is buffered into the processor system model wrapper. This is the source data event for the address variable.
- The signaling event “A15-A0 address lines are identified” occurs and triggers the transfer of address variable value from the processor wrapper buffer to the memory wrapper buffer. This is the communication data event of the address variable.
- The D7-D0 data lines are identified. The value of address variable is transferred to the memory target component. This is the destination data event of the address variable.
- The memory target component executes a read atomic event and returns the contents of the memory location identified by the address variable. This value is transferred from the memory target component to the memory wrapper buffer. This is the source data event of the contents variable.
- The falling clock edge, after the D7-D0 data lines are identified, triggers the transfer of contents variable value from the memory wrapper buffer to the processor wrapper buffer. This is the communication data event of the contents variable.
- A falling clock edge occurs and satisfies the timing constraints of the “Time S is Low” (TSO) AC-Characteristic. It triggers the transfer of the contents variable value from the processor wrapper buffer to the processor target component. This is the destination data event of the contents variable. This falling clock edge represents the end of the read protocol.

Figure 4-27: Dataflow Presentation Layer

4-6-2 Segment Data Structure

Figure 4-28 shows the required Dataflow XML files in the protocol model generation. The model factory needs four XML files to instantiate Dataflow data structures: the protocol Signature's file (for protocol signature information), the Data Event's file (for protocol Data Events information), the Paths file (for tracking signature variables) and protocol Flags file (since every data event requires a triggering flag).

Figure 4-28: Dataflow XML Inputs in Protocol Model Generation

Figure 4-29 shows the Dataflow segment UML class diagram. The class diagram is interpreted as follows. The Protocol has a Protocol Signature. The Protocol Signature tracks a set of Signature Variables. Each Signature Variable has a Path of ordered Data Events. The Data Event is triggered by a Flag.

Figure 4-29: Dataflow Package UML Class Diagram

Chapter 5: Phase II: System Model

The previous chapter introduces the first phase in the construction of an HLA-compliant system simulator, in which timing diagrams are broken down into behavior scenarios called protocols. Each protocol is then augmented with user input and transformed into an individual protocol model. The model comprises all logic and data entities required for system model wrappers to realize the protocol. Phase I is the only phase that involves user interaction. At the conclusion of Phase I, a set of protocol models have been developed that collectively address all behavior scenarios described by the timing diagrams and user specifications.

This chapter describes Phase II of the process, where a set of protocol models from phase I are integrated to produce a system model. The system model provides the structure and behavior necessary for the targeted components to interact using all of the protocols.

Figure 5-1 shows where the system model fits in the system simulator architecture. The system model is totally encapsulated by the Component Interface and the Communication Interface. The Component Interface is responsible for all interactions with its associated target component, and hides the system model from the target component. The system model interacts with the target component through the Component Interface. These interactions either initiate a protocol or reflect callbacks during a protocol. The Communication Interface is responsible for communication interactions among federates. The interface hides the system model from the communication platform harness, and separates concerns regarding the communication messaging mechanism. The mechanism

is responsible for transporting arguments and return values among the protocol components across the communication platform.

The system model involves three types of simulation events: target component atomic events, signal events and protocol events. The target component atomic events are enumerated at the Component Interface and implemented by the system model. The system model refines the atomic events into a sequence of signal events, which are executed in the simulation of a protocol model. The protocol events are the allowable forms of interactions between system models via a simulation platform. Phase III introduces a fourth type of simulation events which are simulation platform specific. They wrap the protocol events into messages compatible with the simulation platform.

The system model construction process is carried out by a system model factory (figure 5-2). The system model factory accepts system protocol models as inputs and generates their system model classes. The generated code is produced dynamically in an iterative process. Although the implementation of the system model factory is not included in the thesis scope, the process of constructing the system model modules is discussed in this chapter.

This chapter also explains the system model elements by illustrating their structures, important implementation parts and functionalities. A simplified dataflow example is elaborated to highlight the interactions in three zones:

- Between the Component Interface and its system model
- Among system model internal entities
- Between the system model and the Communication Interface.

Phases I and II are independent of the runtime environment, and the results of these phases could be reused, without modification, in a variety of environments. The next chapter outlines phase III in which an HLA-specific interface harness is built for the HLA RTI environment.

Figure 5-1: The System Model in System Simulator Architecture

5-1 System Model Construction

The system model factory is a code generator, which processes the input protocol models to create a corresponding system model wrapper as shown in figure 5-2. The process of

constructing a system model wrapper does not depend on the order in which protocols are input.

Figure 5-2: Constructing a System Model Wrapper

Figure 5-3 shows the elements of the system model factory. As a code generator, its functionality is to create and extend system model wrapper classes. The system model factory has two main building blocks: the factory manager and the stream writer. The stream writer contains all IO utilities required for reading or writing files. The factory manager contains the logic and handlers for different code generation cases. The implementation of the system model factory is not included in this thesis scope but the implemented example provides the generated code under the assumption that a system model factory exists. The process of generating a system model wrapper is discussed later in section 5.4. The code generated by the system model factory is combined with the input protocol models to form the system model wrapper.

Figure 5-3: System Model Factory

5-2 System Model Elements

Figure 5-4 shows the system model elements, consisting of a Command tree (Command super-class and its sub-classes), an Agent tree (Agent super-class and its sub-classes), a Simulator, the Agent Table and Command Generator accessories, and the contributing protocol models.

Figure 5-4: System Model Elements

Figure 5-5 shows the elements in the context of a simplified UML class diagram. Note that the Communication Interface interacts with the system model through both the Agent and Simulator.

Figure 5-5: System Model Simplified UML Class Diagram

System model elements cooperatively preserve the logic and data required to represent all of the integrated protocols. In the remainder of this section, the system model elements are elaborated in terms of the role of each element during a simulation, its structure, and implementation rationale.

5-2-1 Simulator Element

The role of the component Simulator is to control the initialization of each instance of a protocol model and to clean up after the performance of the protocol model simulation. When a target component triggers a protocol model, the component's Simulator receives the call through the Component Interface. The Simulator consequentially notifies the Simulators in all other participating federates by broadcasting the beginning of the protocol simulation. Each participating Simulator instantiates the required entities for performing the protocol simulation at its federate. After the protocol simulation is performed, each Simulator cleans up by deleting local instances which are no longer of use. Figure 5-5 contains a UML class diagram which shows where the component Simulator fits in the system model wrapper. Each system model has only one unique Simulator, which handles all target component interactions.

5-2-2 Agent Element

The Agent super-class consists of the general attributes and state variables common to all system models. Each model has a concrete Agent sub-class, which represents its specialization. The role of a concrete Agent includes:

- **Accessing protocol models.**

An Agent must track all instances of its constituent protocol models. The Agent has functionality for accessing and filtering data from the protocols.

- **Interactions with Command module**

Each protocol tracked by the Agent has a Command concrete class, which encodes the role of the component in the protocol. The Command element is discussed later in

section 5-2-3. The Agent has a collection of all of its associated commands. An Agent can retrieve a command by its protocol name. The Agent initializes commands and manipulates interactions from and to commands.

- **Broadcasting and receiving updates**

The Agent works as a key element of flow control by dispatching protocol progress updates, in the form of protocol events, through the Communication Interface. The Communication Interface transfers the updates through the communication platform, to participating Agents. The Agents that receive the updates then react to the event locally.

- **Implementing protocol interfaces (header files).**

The protocol interface represents the target component atomic events. When constructing a system model and a protocol is introduced, the Agent super class is extended to include the introduced protocol interface. The super-class has an empty implementation for the introduced protocol interfaces, and complete implementations are included in the associated concrete Agent sub-classes. This implementation enables the use of polymorphism.

Structure

A tree of concrete Agent sub-classes is constructed dynamically during the system model building process, and finally, a concrete Agent is mapped to each individual target component. Figure 5-6 shows the UML class diagram of the implementation example consisting of “Processor” and “Memory” components. In the example there are two different protocol models: “read” and “write”. The Agent super-class implements the

interface of each protocol model. Figure 5-6 shows that the concrete classes ProcessorAgent and MemoryAgent both inherit from Agent

Figure 5-6: Example of Agent Tree UML Class Diagram

5-2-3 Command Element

The goal of the concrete Command sub-class is to perform the role of a component in the process of simulating a protocol model. A component's role in a protocol is a time-ordered sequence of timing events. Each Command filters the component's role at runtime, and performs the protocol simulation by executing the signaling events that it is responsible for. The execution of signaling events includes generating protocol events that are discharged through the Agent and dispatched to other participating federates

through the communication platform. When protocol events are received at participating federates, they are reflected to the appropriate Command object. The subscriber component Commands are reflected by the received protocol events as a part of simulation interactions. The characteristics of the Command class depend on the pair {component, protocol} to identify a unique concrete Command. The Command super-class has an abstract method called execute which all Command sub-classes must implement. The execute method is responsible for the start of the Command's thread of execution.

Structure

Figures 5-7 shows a Command tree UML class diagram that includes Commands for the read and write protocols. There are two concrete command classes associated with the Memory component: MemoryReadCommand and MemoryWriteCommand. Each of them inherits from Command super class to represent roles in the Memory component's system model. For example, the MemoryWriteCommand encodes the Memory component's role in a write protocol.

Figure 5-7: Example of Command Tree UML Class Diagram Implementation

Command is a super-class, which inherits from Thread. In a protocol, each component has a representative concrete Command subclass. This means that for a protocol, with say four target components, four different Command subclasses are needed to simulate the protocol model of the system. In the implementation example in figure 5-7, the names of the Command sub-classes begin with the component name followed by the protocol name

and then “Command”. For example: MemoryReadCommand implements the read command for the memory component.

The system model provides each component with all the ordered events associated with a command. Each component is responsible for filtering its own signaling events to execute. Figure 5-8 shows the Command run method, which is responsible for filtering and processing the component signaling events along the simulation progress. The implementation of this method requires the leader component to process the event first. Once the leader finishes with the event, the leader waits for other components to acknowledge that they have processed the same event.

```
public void run
{
    For EACH signaling event in the protocol timed ordered events
    sequence
    {
        IF this system model is not the LEADER
            WAIT for a signal mapped to current event
        Process current event
        IF this system model is the LEADER
        {
            • Echo a signal mapped to current event to allow other
              system models to process current event
            • WAIT until all associated system models finish
              processing the current event
        }
        ELSE
        {
            Signal LEADER, this system model finished
            processing current event
        }
    } //END OF ALL EVENTS
    Broadcast an end of command
    STOP the Thread
}
```

Figure 5-8: The run Method in the Command Thread

Figure 5-9 shows the Command class method, which is responsible for processing the current signaling event. If the signaling event is controlled by the current system model, or is a shared clock event, the AC-characteristics and flags associated with this event are updated. If the current signaling event is specified as an end of the protocol state, the protocol state is updated.

```
void processEvent(event)
{
    IF the event is controlled by this system model, or it is a clock event
    {
        • Update the AC-Characteristics of the current protocol state by the
          current event.
        • Update protocol flags by the current event.
        • Update protocol flags by the newly generated AC-Characteristics.
    }
    IF the current event is an end of protocol state event
    {
        Update current state to next state
    }
}
```

Figure 5-9: Event Processing

Processing an event may generate flags. The implementation uses flags for exchanging progress measures among components. The generated flags are wrapped in protocol events, dispatched to other federates, and recorded in each participating system model. Each system model checks the consequences of the newly generated flag. The following method shown in figure 5-10 is responsible for updating protocol flags by an event.

The protocol flags associated with the event are extracted in a collection of candidate flags. Each candidate flag has a list of prerequisite flags to ensure the correctness of the protocol progress. If these prerequisite flags are achieved during the protocol progress,

then the flag is successfully generated and its value becomes TRUE. The newly generated flag is added to the list of runtime-achieved flags and broadcasted to all system model wrappers. If the prerequisites list of flags is not fulfilled, an exception is raised.

```
void update_protocol_flags_by_current_event(anEvent)
{
    Extract the set of protocol flags which is associated with the current
    event
    For each flag in the event associated set of flags
    {
        Get the prerequisite set of flags to achieve the current flag
        IF the entire prerequisite set of flags has been achieved in the
        runtime at the local component or at any other component
        {
            • Approve the current flag.
            • Add it to the runtime achieved flags of the
              current component
            • Update all components by the newly generated
              flag where:
              1. It is added to the newly achieved flags
                in each component runtime flag
                container.
              2. The consequences of the occurrence of
                such flag are processed.
        }
        Else
        {
            Exception takes place
        }
    }
}
```

Figure 5-10: Updating Protocol Flags by a Current Event

The flag may trigger a dataflow (data event) of a protocol signature data value. The `updateSignatureVariable` method shown in figure 5-11 is responsible for processing a data event due to the occurrence of its triggering flag. The processing of the data event

depends on its type. If the data event is a source data event, the signature variable value goes from the source target component to be buffered into the source system model. If the data event is a communication data event, the value is dispatched from the source system model to all participating system models. At the end of executing the communication data event, the signature variable value resides in the receiver system model buffer. If the data event is a destination data event, the signature variable value transfers from the system model buffer to its destination target component.

```
void updateSignatureVariable()
{
1-get the signature variable current data event.
2-execute the data event:
    Case:
        Current data event is a source data event.
        {
            The value is buffered from the target component into the
            system model wrapper.
        }
    Case:
        Current data event is a communication data event.
        {
            Broadcast the variable value to all associated system models.
            The value is stored at the receiver system model wrapper
            buffer.
        }
    Case:
        Current data event is a destination data event.
        {
            The value transfers from the system model wrapper buffer to
            its destination target component.
        }
}
```

Figure 5-11: The Processing of a Data Event

5-3 Wrapping the System Model

Figure 5-1 and figure 5-5 shows that the system model is wrapped by the Component Interface and the Communication Interface to form the system model wrapper. Both of the interfaces encapsulate the system model functionality establishing a separation of concerns. The Component Interface deals with all behavior atomic events while the Communication Interface handles all protocol events.

Wrapping the system model ensures a fully independent design and implementation of the system model. Extending the system model does not affect other elements as long as the wrapping interfaces are respected. Wrapping the system model allows the reuse of the model above different platforms. The use of the system model above a platform requires a communication harness to mediate between the system model wrapper and the platform.

The Communication Interface introduces a set of rules to allow system models to communicate and coordinate activities through the communication platform (HLA, RMI, CORBA, MPI). Communications among system models take place by exchanging protocol event messages. The messages can carry parameters and return values across the communication platform. All required calls to perform communications are enumerated. Both the system model and the communication platform harness recognize these enumerated communication calls.

Establishing a message passing mechanism relies on the communication platform harness. If the communication platform provides a message passing (MP) mechanism, then the communication platform harness employs the ready-made platform MP mechanism. If the communication platform does not provide such MP services, then its

corresponding harness must implement the functionality. The message passing mechanism is contracted by the system wrapper, and the Communication Interface enumerates the entire message passing.

5-4 Tool Specifications of Creating a System Model

As previously mentioned in section 5-1, the thesis research does not provide an implementation for the system model factory but it provides specifications of the system model construction. The implementation initially provides the Command super-class and the Agent super-class. The other system model classes are assumed to be generated dynamically in the process of building the system model. Figure 5-12 demonstrates a proposed flowchart to describe the iterative process of building a system model out of its individual protocol models as follows.

For each newly introduced protocol model, the code of the Agent super-class is altered to include the protocol model interface. Then, for each component involved in the newly introduced protocol model, if the system model does not have an associated Agent sub class to represent the component, the tool (system model factory) creates a new concrete Agent sub-class that represents the component and creates a new concrete Command sub class to represent the component's role in this protocol. The tool updates the Agents table by a new record of (component name, Agent sub class instance) unless the record exists already; and updates the Command Generator by the record (component name, protocol name, Command sub-class instance).

Figure 5-12: Tool Specifications for Generating the System Model

5-5 System Model Data Flow

The dataflow of a system model involves three different zones of study. The first zone is the dataflow between the system model and the Component Interface where system calls (atomic events) take place. At this zone, a call may take place from the leader Component Interface to the system model to initiate a protocol. In another scenario, a callback atomic event may take place from the system model to the Component Interface. The second zone is the dataflow among the internal entities of the system model where internal

interactions occur to perform the component role of involvement in the invoked protocol. The third zone is between the system model and the Communication Interface where the updates and callbacks take place among participating components.

Figure 9 shows a simplified example of write protocol dataflow diagram. The write protocol is initiated by the Processor component with the participation of the Memory component. Initially the Processor Component Interface passes the write atomic event to the Processor Simulator. The Processor Simulator notifies the Memory Simulator of the beginning of the write protocol. Consequently the Processor Simulator initializes the ProcessorWriteCommand and the Processor Agent, while the Memory Simulator initializes the MemoryWriteCommand and the Memory Agent. Each Agent invokes its protocol Command to start its command thread of execution. Each Command refines the behavior atomic event into a relevant sequence of signal events. The Commands execute the signal events and broadcast progress updates to participating Agents in the form of protocol events. Updates are exchanged through Communication Interfaces. As shown in figure 5-13, the simplified write protocol dataflow ends when the MemoryAgent fires a write callback atomic event through the Memory Simulator. The Memory Simulator dispatches the call to the Memory Component Interface and starts the cleanup process.

Figure 5-13: System Model Dataflow Example

Chapter 6: Phase III HLA Harness

Previous chapters introduce phase I and phase II of constructing an HLA-compliant system simulator. In phase I, the system behaviors are represented as protocol scenarios specified in timing diagrams. Each protocol scenario is refined (decomposed) into its signal interactions, and augmented with user specifications to form an individual protocol model. At the conclusion of phase I, a collection of protocol models are developed to address all the possible protocol scenarios of the system. In phase II, the protocol models from phase I are collectively integrated to form a system model wrapper, which encapsulates the logic and data necessary to simulate the protocol scenarios. This chapter introduces phase III (the final phase) in constructing the HLA-complaint system simulator. The chapter focuses on the construction and functionality of an HLA harness. The main goal of the HLA harness is to allow the system model generated in phase II to participate in a federation above the HLA RTI.

The HLA harness is designed to mediate between the system model wrapper and the HLA simulation platform. It utilizes the distributed HLA services to allow interactions and synchronization among federates. The Harness provides a message passing mechanism to enable publishing and subscribing federates to exchange protocol events. Through out the message passing mechanism, publishing federates produce protocol events at the system model wrapper and pass them on to the HLA harness for dispatching. The harness translates the protocol events into HLA-specific update events to be carried out by the HLA platform. Depending on the settings and configurations of the HLA platform, the platform triggers subscriber federates by HLA callback

reflections. The subscriber harness receives the HLA reflections, translates them back into protocol events, and discharges them to be processed by its system model wrapper.

As shown in figure 6-1, the HLA Harness consists of three main blocks: the Harness Manager, the Service Manager and the Message Passing (MP) Manager. The main responsibilities of the Harness Manager is to configure the necessary HLA settings, policies and strategies of the federate and federation to allow their activities from the creation to the destruction. The MP Manager and the Service Manager cooperate to allow federate interactions by exchanging protocol events via the HLA RTI. The MP Manager hides the system model wrapper while the Service Manager hides the HLA platform. In addition, the Service Manager provides a time advance mechanism, which allows the federation to synchronize publish and subscribe events. The federate logical time does not correspond to wall clock time. The time advance depends on the process of producing and consuming events within the federation.

Figure 6-1:HLA Harness Architecture

The location of the Harness in the architecture of the simulator dictates its construction, as the HLA harness is restricted by the HLA platform and the Communication Interface of the system model wrapper. The Communication Interface contracts the communications (publish or subscribe events) between the system model and the HLA harness. The Harness event exchange mechanism must honor the Communication Interface requirements. Also, the HLA platform explicitly contracts the allowable management services and discipline. Any implementation of the HLA harness must follow the HLA rules.

In this chapter, section 6-1 explains the message passing mechanism in details. The section shows how to build an MP Manager from an existing Communication Interface and how it functions. An example is provided in subsection 6-1-2 to demonstrate the MP dynamics. Section 6-2 introduces the Service Manager and its components. The section explains the role of each component and how the Service Manager utilizes the HLA distributed services to exchange protocol events. The section also introduces a federation progress approach, based on publish and subscribe events in a two-federate federation. Section 6-3 demonstrates some runtime scenarios to allow better understanding of how the HLA is utilized. The section concludes by introducing a progress approach that is not restricted to only two federates.

6-1 Harness Message Passing (MP) Mechanism

The goal of introducing a system-specific HLA MP mechanism is to allow federates to exchange protocol events using the HLA RTI as a simulation platform. The HLA

platform does not explicitly support an MP mechanism to convey a protocol event among federates. Instead, the platform provides two forms of federate information exchange: object instances and interaction classes. Either form could be used to build an MP mechanism. HLA interaction classes can allow an MP mechanism by mapping each protocol event method call to an interaction class. This MP implementation may require many interaction classes. On the other hand, the MP mechanism can be implemented by using a set of object classes that can form any protocol event. Although this approach has more flexibility and extensibility, object classes describe types of things that can persist, and persistency is not required to enable an MP mechanism. The approach taken in this research is to use object instances to implement the MP mechanism, since it offers more reusability, and avoids the requirements for many interaction classes.

In this chapter, the MP mechanism is established using an MP Manager, which is separated from the HLA platform behind the Service Manager. It receives messages from the system model wrapper, codes them into records of parameters, and delegates them to the Service Manager. It also accepts deliveries from the Service Manager in the form of coded records, and decodes them back into system wrapper messages. The resulting MP mechanism is system specific. It depends on the published and subscribed messages intended for each federate in the system.

6-1-1 Building the MP Manager

Figure 6-2 shows the elements of the MP Manager. The MP Manager itself consists of two tables: Code Table and Decode Table. The Code Table receives a protocol event for

publishing from the system model wrapper and codes it into a record of parameters. The Decode Table works in reverse, by receiving a record of parameters and decoding it to a protocol event (subscribed event). The MP implementation depends on the existence of the Communication Interface, which works as a contractor for the federate's outgoing events and incoming reflections. The publisher remote interface is the set of method calls that allows a federate to dispatch protocol events from the system model wrapper to subscribing federates. The subscriber remote interface is the set of reflection calls that allows a federate to receive every protocol event of interest to its system model wrapper. The publishing and subscribing remote interface determine the publishing and subscribing domains. The domains are necessary to set each federate's configuration in the .FED file, which declares them in the runtime settings of federates.

A federate Communication Interface contains its publisher remote interface and its subscriber remote interface. For each federate, each method in the publisher interface maps to a subscriber remote interface reflection in at least one other federate. If there are multiple federates which subscribe to the same method, then a one to many mapping exists. The reflection method is listed in the subscriber remote interface of each subscribing federate. The publisher call broadcasts the required values to represent a protocol event, while its mapped reflection method executes the protocol event at the subscriber federate (precisely at the federate system model wrapper). To distinguish the calls, an enumerated method identifier is assigned to each method. The publisher interface call and the associated subscriber reflections have the same method identifier. The direction of the event decides if it is a call event or a reflection event.

The Code and Decode tables work together to simplify message passing. The Federate code table encodes each publisher remote interface call into a call record of parameters for communication through HLA RTI services. The code table headers are the minimal required parameters to represent any message that will be sent. The key field of a call record contains the enumerated method identifier for the encoded call. The rest of the record values represent parameters to the call. If the call does not need all the headers to represent its parameters, the call record is compressed to contain the method identifier besides only the required number of parameters. The federate decode table decodes a call record into a subscriber remote interface call. The code table is necessary in the process of translating protocol events into HLA specific events. The decode table is necessary in the process of translating HLA reflection events into protocol events.

Figure 6-2: MP Manager Design Elements

A federate's publishing domain is the set of variables published via the RTI that allows the federate to publish call records. The federate can change any variable of the publishing domain, and update the federation the new values. The federate publishing domain is comprised of the federate code table headers. The federate subscribing domain is the set of variables subscribed via the RTI that allows the federate to receive call record update reflections. The federate's subscribing domain is comprised of the federate decode table headers.

6-1-2 MP Dynamics Example

Figure 6-3 illustrates the exchange of a protocol event between a sender federate and a receiver federate. The sender federate's publisher remote interface enumerates seven calls in its code table. The receiver federate mirrors the calls in its subscriber remote interface decode table. The following describes the communications shown in figure 6-3, in which a call to method 5 is exchanged:

- 1- Before any events are exchanged, the sender federate must declare the set {Call Identifier, W,X,Y,Z} (i.e. the code table header) as a published set of variables, while the receiver federate must declare the same set as the subscribed set of variables.
- 2- The system model wrapper starts the exchange by generating the protocol event (5,Z,X) which corresponds to the method 5 call. The event is delegated to the HLA Harness.

- 3- The HLA Harness uses code table information to encode the event in the appropriate call record. The record contains the method identifier as a key and the call parameters. The record specifies new values for a subset of the published set of variables {Call Identifier, X, Z}.
- 4- The HLA Harness generates an HLA update event for the selected variables. See HLA event Update {Method Identifier =5,Z,X}.
- 5- Since the update involves variables to which the receiver subscribes, a reflection event is generated from the HLA to the receiver's HLA Harness. See the HLA event: Reflect {5,Z,X}.
- 6- The receiver's harness extracts the wrapped record and decodes it into the reflection method that represents the original protocol event. The decode table is used to resolve the reflection method and its parameters from the conveyed information.
- 7- The receiver HLA Harness delegates the reflected protocol event to its federate system model wrapper. See the reflected Protocol event {5,Z,X}
- 8- The protocol event is executed at the receiver side by invoking the method mapped to the appropriate entity in the subscriber remote interface. See Reflect#5 {Z,X}.

Figure 6-3: HLA-MP Mechanism

6-2 Service Manager

The goal of the HLA Service Manager is to utilize the HLA RTI services to allow federates to interact while processing through a protocol. The HLA Object Management services are used to control the generation of update events and the release of the callback reflections. The HLA Time Management services are used to synchronize simulation

progress. As shown in figure 6-4, the Service Manager is comprised of three controllers: Production Controller, Consumption Controller and Progress Controller. The Production Controller is responsible for update events, and the Consumption Controller is responsible for receiving reflections. The Progress Controller manages time advance to allow synchronization among federates. The following paragraphs of this section explain the role of each controller, while section 6-3 presents a set of runtime scenarios to demonstrate simulation dynamics.

Figure 6-4: HLA Service Manager

6-2-1 Production Controller

The Production Controller manages the process of updating the federation of the occurrence of protocol events. To synchronize the updates, the process obligates the federate to generate Time Stamp Ordered (TSO) HLA update events. The controller receives the MP record from the MP Manager and extracts its contained parameters. As described in 6-1, the record key (method identifier) directs the controller to extract the proper set of parameters out of the record fields. The controller decides a time stamp for the current incoming record, which equals the current federate logical time plus one unit of time (i.e. a simulation step). Other records wait until the current record is updated and released. The controller generates an update event by passing the extracted parameters and time stamp to the `updateAttributeValues()` method provided by the HLA RTI Object Management Services.

6-2-2 Consumption Controller

The Consumption Controller manages the process of receiving TSO HLA reflection events and translating them into protocol event records. The process obligates the federate to be able to distinguish TSO events. The HLA RTI platform involves the Consumption Controller of the reflection through the callback Object Management Service `reflectAttributeValues()`. The controller extracts the set method identifier and the set of parameters out of the reflection event, and constructs protocol event record. Finally, the Consumption Controller delegates the reformed record to the MP Manager.

6-2-3 Progress Controller

The Progress Controller manages the time advance, which allows the production and consumption of federate TSO events, and synchronizes logical time among federates. The Progress Controller's contribution could be summarized in the following roles:

- It senses the incoming event record from the MP Manager and consequently signals the Production Controller to process the incoming record.
- On the publisher federate side, once the Production Controller generates a TSO event, the Progress Controller requests a time advance of the current publisher federate logical time to match the time stamp of the generated event.
- On the subscriber federate side, once the Progress Controller senses a TSO event waiting for release, it requests a time advance of the current subscriber federate logical time to match the time stamp of the waiting event.

6-2-4 Two-Federate Approach to Time Advance

This section introduces an approach to control federation time advance and the progress of the simulation. This approach works only for two federates. A modified multiple-federate approach, which works for multiple federates, is discussed later. Both of the approaches assume that all federates are always synchronized at the same point of logical

time, and that all of the federates are capable of generating and distinguishing TSO events.

Figure 6-5 shows the process of consuming and producing TSO events in a federation of only two federates. The figure is further explained in the following paragraphs.

For a publisher federate, if the Progress Controller senses an incoming event record from the MP Manager, it consequently signals the Production Controller to process the incoming record.

Line 1

- The Production Controller decides a time stamp = current federate logical time + one time unit ahead (simulation step).
- It translates the record to generate a TSO HLA event with the decided time stamp. The TSO event interpretation is a promise to broadcast an event at a future federate logical time equal to the current time plus the simulation step.

Line 2

- Once the Production Controller generates the TSO event, the Progress Controller requests to advance the logical time of the current federate to the time of that event.

Line 3

- The publisher federate waits until the HLA RTI platform agrees to grant a time advance.

Line 4

- The Production Controller is acknowledged by a time advance grant, and the waiting federate is now free for further processing.

For a subscriber federate, if the Progress Controller senses that a TSO event with a time stamp greater than the federate current logical time is waiting for release:

Line 5

- The Progress Controller requests to advance the logical time of the current federate to the time of the waiting TSO event.

Line 6

- The subscriber federate waits until the HLA RTI platform grants the time advance.

Line 7

- The Consumption Controller is acknowledged by a time advance grant, and the waiting federate is now free for further processing.

Figure 6-5: Producing and Consuming TSO Events

6-3 Runtime Scenarios

The goal of demonstrating the runtime scenarios is to provide better understanding of how the HLA RTI is employed to support the exchange of events and simulation

progressing approach. The scenarios illustrate how the HLA RTI is being utilized as a simulation platform, and clarify how federates are integrated to simulate system models.

Case: White Federate Produces an Event

In the scenario in figure 6-6, the behavior involves the white federate and the green federate. This scenario captures a case where the white federate has an event to publish, while the green federate does not. The scenario demonstrates the following sequence of actions:

- 1- The white federate updates an event (promises an event) with a time stamp $T_{promise} = \text{current federate logical time} + \text{simulation step}$. This promise causes the lower bound time stamp of the green federate to be at time $T_{promise}$ since the green federate is not expecting any other event before this time.
- 2- The white federate requests the advance of its logical time to $T_{promise}$, but this request is held by the HLA RTI as white's lower bound time stamp is at $T_{current}$ which is less than $T_{promise}$. The lower bound time stamp is at $T_{current}$ since the green federate can issue an event at a time greater than or equal to $T_{current}$ but less than $T_{promise}$.
- 3- The green federate requests the advance of its logical time to $T_{promise}$.
- 4- Since the green federate's lower bound time stamp is greater than or equal to the requested time, then green federate is allowed to advance its logical time to $T_{promise}$.

- 5- Now, the minimum time that the green federate can broadcast an event is not less than $T_{promise}$. This increases the lower bound time stamp of the white federate to $T_{promise}$.
- 6- As a consequence, the white federate's early advance request is granted and its logical time is advanced to $T_{promise}$.
- 7- Progress is achieved in this scenario, and the green federate is able to consume the update event since its logical time matches the time stamp of the event.

Figure 6-6: Only One Federate Produces an Event

Case: Each Federate Produces an Event at the Same Time

This scenario, shown in figure 6-7, captures a case where both the white federate and the green federate have an event to dispatch at the same time. The scenario demonstrates the following sequence of actions:

- 1- The white federate promises to broadcast an event at a time $T_{promise}$.
- 2- Such promise causes the lower bound time stamp of the green federate to be at a time equal to $T_{promise}$ because the green federate does not expect any other event before this time.
- 3- The white federate requests an advance of its logical time to $T_{promise}$. This request is held by the HLA RTI as white's lower bound time stamp is at $T_{current}$ which is less than $T_{promise}$. Up to this moment the green federate can issue an event at a time greater than or equal to $T_{current}$ and less than $T_{promise}$.
- 4- The green federate promises to broadcast an event at a time $T_{promise}$. This promise causes the lower bound time stamp of the white federate to be at a time equal to $T_{promise}$.
- 5- As a consequence, the white federate is allowed to increase its logical time to $T_{promise}$, which is less than or equal to the white federate's lower bound time stamp.
- 6- The green federate requests an advance of its logical time to $T_{promise}$.
- 7- The green federate request is accepted since the requested time is less than or equal to its lower bound time stamp.
- 8- Progress is achieved in this scenario, and both of the events are released and reflected to the subscriber federates.

Figure 6-7: Each Federate Produces an Event

Case: No Production of Events

Figure 6-8 shows a scenario where there are no promises of any updates and there are no requests for any time advances. The simulation does not progress, but it is accepted because there is no denial of services. There is no published event that could not be consumed by a subscriber federate.

Figure 6-8: No Production of Events

Advancing Time in a Federation of More than Two Federates

In the hypothetical scenario shown in figure 6-9, the behavior involves more than two federates (F1, F2 and F3 federates). This scenario captures a case where the two-federate time advance approach fails. The failure case occurs when a federate updates an event while at least one federate is not interested in such event.

In figure 6-9, the scenario demonstrates the following sequence of actions:

- 1- Federate F1 promises to broadcast an event at a time $T_{promise}$.
- 2- This promise is not enough to cause the lower bound time stamp of federate F2 or federate F3 to increase. F2 may broadcast an event at a time greater than or equal $T_{current}$ and less than or equal $T_{promise}$, so F3's lower bound time stamp is still

$T_{current}$. F2's lower bound time stamp is also $T_{current}$ as the possibility exists that F3 may broadcast an event at a time greater than or equal to $T_{current}$ and less than or equal to $T_{promise}$. The same is applied on F1 lower time stamp.

- 3- F1 requests an advance of its logical time to $T_{promise}$, but this request is held by the HLA RTI as F1's as its lower bound time stamp is at $T_{current}$ (which is less than $T_{promise}$).
- 4- F2 requests an advance of its logical time to $T_{promise}$, but this request is also held by the HLA RTI.
- 5- F3 asks to advance of its logical time to $T_{promise}$, but this request is also held by the HLA RTI.

The three federates are trapped in a deadlock where each federate is waiting for another federate to advance. The simulation cannot progress in this scenario. There is an update event produced by F1, yet both F2 and F3 cannot consume the event.

Figure 6-9:Deadlock Scenario

Figure 6-10 shows the multiple-federate progress approach. It is the same as the two-federate approach except for the case where a federate does not have any record to publish; it just pumps a fake event with a time stamp equals $[T_{\text{current}} + \text{simulation step}]$. This solves the deadlock problem that arises in the previous scenario, but it has the overhead of pumping unnecessary fake events.

Figure 6-10: Advancing Time in a Federation of more than Two Federates

Chapter 7: Conclusions

7-1 Conclusions

The following conclusions are derived from this research.

- 1- An approach is proposed to model and simulate hardware systems. The approach addresses the signaling scenarios of system behaviors. The signaling scenarios are specified using timing diagrams.
 - ❑ Hardware models which discount the signaling scenarios, are reused and extended to model the system behaviors at the level of the signaling scenarios. The atomic behaviors of the existing models are refined into sequences of signaling events.
 - ❑ Behavior-interfacing scenarios are modeled using timing diagram specifications. The timing diagrams are represented using TDML notation, which is an XML-based format.
- 2- An HLA-framework is defined to allow the HLA compatibility of the proposed system model simulations.
 - ❑ A message passing mechanism is proposed to transmit protocol events across the HLA RTI. The mechanism identifies the published and subscribed object domains needed to convey published events to the subscribing federates.

- ❑ Service controllers are implemented to manage the federation and federate activities. The service controllers employ the HLA RTI's support in regulating the federation and federate activities (e.g. the creation of the federation, joining the federation, resigning the federation, publishing protocol events, consuming subscribed events, etc.).
 - ❑ A federation progress approach (dead lock free) is implemented. A restricted HLA RTI time management policy is implemented. Each federate is restricted to be both time regulating and time constrained. Each federate is fully involved in time management. The policy synchronizes the production of published events and the consumption of subscribed events. The approach allows federation progress and protects against deadlocks.
- 3- The proposed approach emphasizes reusability, scalability and interoperability.
- ❑ The design decisions underlying the approach promote the use of XML extensively in the modeling process. XML standards support interoperability in a heterogeneous distributed computing environment. Also, the choice of the HLA as the simulation platform supports the simulation interoperability goals.
 - ❑ Object Oriented design for reusability and scalability is implemented. A model architecture is provided, which considers the development of object-oriented simulations. The architecture components are loosely coupled. The system model is wrapped by two contracting interfaces. The system model wrapper is integrated with an HLA harness to provide an HLA compatible simulation. Simulation reusability and scalability is achieved in several forms:

- 1- The model architecture reuses and extends existing hardware models by refining the abstract atomic behaviors into sequences of signaling events.
- 2- Integrating the system model wrapper with an appropriate platform harness; enables the reuse of the system model across platforms.
- 3- The proposed segment-based approach in constructing the protocol models allows the extendibility of the models. Extending the protocol model by introducing a new segment does not require any major alterations to other segments.
- 4- A support tool is proposed to generate an HLA Compliant System Simulator. The simulator is constructed in three consecutive phases. The tool automates the generation of the simulation modules needed to include signaling behaviors. The tool assumes that timing diagrams are specified in TDML. A detailed specification of this tool is part of this work.

7-2 Future Research

Future work for this research includes the dynamic code generation of the system model, the development of multi layer GUI for protocol model data collection, and the enhancement of a migration methodology from a non-HLA simulation to an HLA simulation.

7-2-1 Dynamic Code Generation

The dynamic code generation of the system model wrapper is achieved by a system model factory. In this research, the system model factory was not implemented. The

system model code and the wrapping interfaces were provided manually in the implementation example. In the future, the system model factory should be implemented to create and extend the system model wrapper classes and interfaces. The implemented system model factory should integrate the system protocol models iteratively where the order of protocol models does not matter.

7-2-2 Multi Layer GUI for Protocol Model Data Collection

Chapter 4 introduces a segment-based approach in the protocol model construction. Each segment is constructed in two steps: A data collection step and an initialization step. The data collection step ends by generating XML-based files, which encodes the segment's data records. This research did not implement a tool to augment timing diagrams with user inputs, and generate the XML representation of the segment data records. The XML files in the implementation example were provided manually.

A multi-layer GUI application tool should be implemented in the future. The tool should allow the user to input and edit segment information. The user should be able to visualize the inputs and save them in an XML format. The tool should also allow the user to retrieve segment files for further modification and/or visualization.

7-2-3 Introducing a Migration Methodology from a non-HLA Simulation to an HLA Simulation

In this research, the HLA harness is integrated with the system model wrapper to enable an HLA system model simulation. The construction of the HLA harness depends on the Communication Interface of the System Model Wrapper and the HLA RTI platform. If a non-HLA simulation has a model of a Communication Interface which contracts all the published and subscribed model interactions, then the migration to an HLA compatible simulation is possible. The simulation migration can take place by constructing an HLA harness using the Communication Interface.

One of the main concerns of the future research is to establish a methodology of detecting a Communication Interface for the existing simulation model. The Communication Interface should contract the published and subscribed events in the federation. The future research should introduce an HLA framework, which satisfies the existing simulation goals and performance. Fulfilling the existing (non-HLA) simulation timing requirements in the HLA migrated version is another challenge.

References

- [1] North Atlantic Council, "NATO Modelling & Simulation Master Plan", version 1, August 1998.
Last Retrieved May 21, 2003, from
<http://www.drdc-rddc.gc.ca/seco/documents/NMSMasterPlan.pdf>
- [2] Anthony Cramp, Michael Oudshoorn, "Employing hierarchical federation communities in the virtual ship architecture", *Proceedings of the twenty-fifth Australasian conference on Computer science* - Volume 4, PP: 41 – 49, January 2002
- [3] Judith Dahmann et al., "A reusable architecture for simulations", ACM Press New York, NY, USA, PP: 79 - 84, September 1999.
- [4] Department of Defense, Defense Modeling and Simulation Office, "RTI 1.3-Next Generation Programmer's Guide", Version 3.2.
Last Retrieved May 21, 2003, from
<http://www.dmsso.mil>
- [5] Merriam-Webster online Dictionary
Last Retrieved May 21, 2003, from
<http://www.m-w.com/>
- [6] Mary Ewing, "Military applications: The economic effects of reusability on distributed simulations", *Proceedings of the 33rd conference on Winter simulation*, PP: 812 - 817, December 2001.
- [7] Ling Feng , Elizabeth Chang , Tharam Dillo, "A Semantic Network-Based Design Methodology for XML Documents", ACM Press New York, NY, USA, PP: 390 – 421, October 2002.
- [8] Mark Fredrickson, Dan Notestein, Melanie Yunk, "A Natural Model for EDA Sensible IP Behavior"
Last Retrieved May 21, 2003, from
http://www.si2.org/si2_publications/ecix/pdf/ip_behavior.pdf
- [9] Richard Fujimoto, "Time Management in the High Level Architecture", College of Computing, Georgia Institute of Technology, Atlanta.
Last Retrieved May 21, 2003, from
http://www.cc.gatech.edu/computing/pads/PAPERS/Time_mgmt_High_Level_Arch.pdf

- [10] Richard Fujimoto, Richard Weatherly, "Time Management in the DoD High Level Architecture", ACM SIGSIM Simulation Digest, *Proceedings of the tenth workshop on Parallel and distributed simulation*, PP: 60 – 67, July 1996.
- [11] Daniel Gajski et al., "Embedded tutorial: essential issues for IP reuse", *Proceedings on the 2000 conference on Asia and South Pacific design automation*, PP: 37 – 42, January 2000
- [12] Telecom Glossary 2000, American National Standard for Telecommunications , American National Standards Institute.
Last Retrieved May 21, 2003, from
<http://www.atis.org/tg2k/>
- [13] John Hamilton, David Nash, Udo Pooch, "Distributed simulation", CRS Press, New York, 1997, ISBN 0-8493-2590-0
- [14] Wen-yang Hsia, "An HLA-based Simulation Environment for Virtual Reality via Java3D", Master thesis submitted to Computer Science and Engineering department, National Sun Yat-sen University, Taiwan, July 2000
- [15] Xueqin Huang, "XML-Based Messaging in the JSIM Simulation Environment", Master Thesis Submitted to the Graduate Faculty of the University of Georgia, Athens, USA, May 2001.
- [16] Ulrich Klein et al., "Traffic Simulation Based On The High Level Architecture", *Proceedings of the 30th conference on Winter simulation*, PP: 1095 – 1104, December 1998.
- [17] Pamela Knight et al., "Evaluation of Run Time Infrastructure (RTI) Implementations". U.S. Army SMDC
Last Retrieved May 21, 2003, from
<https://rsic.redstone.army.mil/hsc/papers/hsc017.pdf>
- [18] Frederick Kuhl, Richard Weatherly, Judith Dahmann, "Creating Computer Simulation Systems: An Introduction to High Level Architecture", Prentice Hall PTR, ISBN 0-13-022511-8.
- [19] Stefan Lenk et al., "Extended Timing Diagrams as a Specification Language", *Proceedings of the conference on European design automation conference*, PP: 28 – 33, September 1994.

- [20] Yew-Hong Leong, William Birmingham, "The Automatic Generation of Bus-Interface Models", *Proceedings of the 29th ACM/IEEE conference on Design automation conference*, PP: 634 – 637, July 1992.
- [21] John Miller, Andrew Seila, Junxiu Tao, "Finding a Substrate for Federated Components on the Web", *Proceedings of the 32nd conference on Winter simulation*, PP: 1849 – 1854, December 2000
- [22] Amir Saghier, "Computer System Modeling at the Hardware Platform Level", thesis submitted to department of systems and computer Engineering, faculty of Engineering, Carleton University, Ottawa, Canada, September 2002.
- [23] Peter Toft, "How to become an HLA guru in a short (er) time", COT/6-6-V1.0, Centre for Object Technology, sponsored by the Danish national center for IT research, Last Retrieved May 21, 2003, from <http://www.cit.dk/COT/reports/reports/Case6/06/cot-6-06.pdf>
- [24] Jeffrey Voas et al., "Software testability: an experiment in measuring simulation reusability", *Proceedings of the 1995 Symposium on Software reusability*, PP: 247 – 255, August 1995.
- [25] Jörg Walter et al., "Hierarchical Random Simulation Approach for the Verification of S/390 CMOS Multiprocessors", *Proceedings of the 34th annual conference on Design automation conference*, PP:89 – 94, June 1997
- [26] Sally White, Richard Reading, "NATO/PfP HLA Federation of VTOL Operations Supporting Simulation Based Acquisition". Last Retrieved May 21, 2003, from http://www.visitech.com/papers/01e-siw-034_nireus.pdf
- [27] Daniel de Freitas Wildt, Flávio Rech Wagner, "Adapting Simulation Environments to HLA: Discussion and Case Study", *Proceedings of the European Simulation Multi-conference*, Prague, Czech Republic, June 2001. Last Retrieved May 21, 2003, from http://www.inf.ufrgs.br/~wildt/dissert/wildtwagner_esm_2001doc.pdf
- [28] George Zobrist, James Leonard, "Object-Oriented Simulation: Reusability, Adaptability, Maintainability", New York: IEEE PRESS, 1997, ISBN 0-7803-1061-6
- [29] Erick Gamma et al., "Design Patterns: Elements of reusable Object-Oriented Software", Addison-Wesley, ISBN 0-201-63361-2