DISCRETE EVENT SYSTEM SPECIFICATION (DEVS)

DISTRIBUTED OBJECT COMPUTING (DOC)

MODELING AND SIMULATION

by

Daryl Ralph Hild

_____

A Dissertation Submitted to the Faculty of the

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfillment of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY

In the Graduate College

THE UNIVERSITY OF ARIZONA

2 0 0 0

THE UNIVERSITY OF ARIZONA ®
GRADUATE COLLEGE

As member of the Final Examination Committee, we certify that we have read the dissertation prepared by Daryl Ralph Hild

entitled Discrete Event System Specification (DEVS) - Distributed Object Computing (DOC) Modeling and Simulation

and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy

Bernard P. Zeigler, Ph.D.	Date
Francois E. Cellier, Ph.D.	Date
Ralph Martinez, Ph.D.	Date
Duane L. Dietrich, Ph.D.	Date
K. Larry Head, Ph.D.	Date

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copy of the dissertation to the Graduate College.

I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

Co-Dissertation Director Bernard P. Zeigler, Ph.D. Co-Dissertation Director Francois E. Cellier, Ph.D.	Date

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the copyright holder.

SIGNED: _____

ACKNOWLEDGMENTS

It hardly seems possible to finally reach the end of this long process, and that it only remains to thank the ones who have helped me out in so many ways, along the way. I know that I will never be able to truly express my appreciation. I can only say: Thank you!

To Hessam, I say thank-you! Thanks for the engaging discussions on software architectures, hardware abstractions, distributed co-design concepts, and distributed systems engineering. You have supported and encouraged me throughout this endeavor and the completion of this dissertation. You have helped make my dissertation the best that I can make it.

To Bernard, I say thank-you! Your insightful and constructive feedback has helped to improve all the work I have done here. Your words of encouragement have helped me reach this objective.

To François, I say thank-you! You challenged me in my efforts to grasp and understand the concepts and theories surrounding modeling and simulation technologies and methodologies. Your feedback and your example have enriched my experience as well as my work. I know that if you like it, every interested audience will be satisfied.

To my wife Janis, my lover, friend, and lifetime companion, I say thank-you! Thank-you for your patience, encouragement, and support in all that I am, all that I do, and all that I pursue. Your support has kept me going. You have helped me keep things in perspective and have helped me keep priorities straight. You make it all worthwhile and I love you so very much.

To my kids, Catherine, Ryan, Naomi, Colby, Anna, Jessica, and Nicholas, I say thank-you! Thanks for being so patient all those times I was "playing" on the computer. And, thanks for being so very ready to play, read, or just enjoy each other when I was through. I love each of you dearly.

Thank you all! — *Daryl*

To Janis,

for her endearing love,

encouraging support,

and lasting friendship . . .

TABLE OF CONTENTS

TABLE OF CONTENTS — Continued

TABLE OF CONTENTS — Continued

LIST OF FIGURES AND TABLES

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

This research examines an approach to modeling and simulating *distributed object computing* (DOC) systems as a set of discrete software components mapped onto a set of networked processing nodes. Our overall modeling approach has clearly separated hardware and software components enabling systems level, distributed co-design engineering. The distributed co-design engineering refers to a formal approach to concurrent hardware and software systems engineering that provides a tractable method for analyzing the inherent complexities that arise in distributed systems. The software abstraction forms a distributed cooperative object (DCO) model to represent interacting software objects. The hardware abstraction forms a loosely coupled network (LCN) model of processing nodes, network gates, and interconnecting communication links. The distribution of DCO software across LCN processors forms an object system mapping (OSM). The OSM provides a sufficient specification to allow simulation investigations. During simulation, the behavioral dynamics of the interacting DCO software components "load" the LCN processing and networking components in terms of memory utilization, computational demands, and communications traffic. The resource constraints of the LCN components, likewise, impose performance constraints on the associated software objects. Class models of the DCO, LCN, and OSM component structures and behavior dynamics were formally developed using the Discrete Event System Specification (DEVS) formalism. These class model specifications were implemented in DEVSJAVA, a Java implementation of DEVS. Class models of

experimental frame components were also developed and implemented to facilitate analysis of individual DCO and LCN components, as well as interdependent system behaviors, during simulations. The resulting DEVS-DOC environment enables distributed systems architects, integration engineers, and automated system designers to conduct performance engineering and trade-off analysis of distributed system structures, topologies, and technologies. This research utilized the resulting DEVS-DOC environment in four case studies. These case studies demonstrate the structural independence and behavioral interdependence of the hardware and software abstractions, the ability to model and simulate real world systems, and the complex interactions that arise in distributed systems can be methodically analyzed.

1. INTRODUCTION

Dramatic increases in both networking speeds and processing power has shifted the computing paradigm from centralized to distributed processing. The economics of an increasing performance-to-cost ratio is continuing to drive this shift with more and more emphasis on building networked processing capabilities. Concurrently, a shift is occurring in the software industry with a move toward object-oriented technologies. This object orientation trend is creating more and more interacting software components. Distributed object computing represents the convergence of these two trends.

This convergence results in a highly complex interaction of the hardware and software components forming the distributed object computing system. Two sets of challenges associated with developing distributed computing systems are inherent complexities and accidental complexities [Sch97]. Inherent complexities are a result of building systems that are distributed. Developers of distributed systems need to resolve issues of distribution; e.g., how to detect and recover from network and host failures; how to minimize communications latency and its impacts; and, how to partition software objects to optimally balance computational loads across host computers with traffic loads across networks. Accidental complexities are a result of inadequate tools and techniques selected to construct distributed systems. Examples of accidental complexity include the use of functional design methodologies; the use of platform specific coding tools; and the use of concurrency and multithreading techniques in coding software components. Functional design methodologies add complexity when attempting to extend or reuse

software components. Platform specific coding tools add complexity when porting code to additional platforms or interfacing components in a heterogeneous environment. Concurrency and multithreading techniques can add unnecessary complexity to software implementation, testing, and maintenance activities.

To attain a level of tractability in developing a distributed object computing system, we advocate the use of modeling and simulation tools to explore design alternatives. We take a "co-design" and synthesis approach in considering the software and hardware implications as well as the communications design issues. In modeling a distributed system, our constructs deal with the partitioning of functionality into software objects, defining software object interactions, distributing software objects across processing nodes, selecting processing components and networking topologies, and structuring communication protocols supporting the software object interactions.

1.1. Motivation

This research is motivated by the growing need for methodologies and mechanisms to support the design and analysis of complex distributed computing systems. Distributed systems approaches are being pursued for a growing variety of business endeavors, including process control and manufacturing, transportation management, military command and control, finance, banking and medical records and imaging. Often, technical conflicts arise in analyzing the user requirements and constraints. For example, military command and control systems may need to exchange large data loads over bandwidth limited networks. A process control system may need

guaranteed job throughputs from remote multi-tasked processors. The complex interactions of such distributed systems make static analysis of behaviors and resource utilization intractable.

To support dynamic analysis, we need a means to generate formal dynamic models of the distributed object computing system of interest. A prime purpose of such dynamic models is the use of simulation to facilitate the needed analysis. In this context, there are three general approaches to system simulation: *emulation, quantum simulation,* and *directed simulation*. *Emulation* refers to highly detailed simulations based on modeling virtually complete system specifications. However, rapidly evolving studies of system alternatives becomes more difficult at this level of specificity. At the other end of the spectrum is *quantum simulation*. Quantum simulation is based on more abstract descriptions (incomplete specification) of system behaviors, which rely on random variables to model *bulk* behavior. Generally, such abstractions enable one to rapidly analyze several alternatives at a distinctly lower level of precision. In between these extremes lies *directed simulation*. Directed simulations focus on modeling a specific aspect of system behavior. Directed simulation requires detailed specifications around the aspect being modeled while ignoring or abstracting away many other aspects.

Within the domain of distributed object computing, commercial products are available to provide directed modeling of the communications network (e,g,, COMNET [CAC99] and OPNET [MIL99]); or, of the software components and their interactions independent of hardware constraints (e.g., WRIGHT [All97]); or, of hardware components via various computer aided engineering tools (e.g., VHDL tools). A

limitation of these available analysis tools, however, is a convenient mechanism to model the software components independent of the hardware components and then explore alternative mappings (distribution) of the software components across the hardware.

The goal of this research is to develop a means of modeling software systems independent of the hardware architecture; modeling hardware architectures as networked computing systems; and coupling these models to form a dynamic system of systems model. Furthermore, this research aspires to have modeling mechanisms that support early-on design analysis using quantum techniques, as well as, enabling specification refinements for more directed analysis of system designs.

1.2. Enterprise Computing

The behavioral complexity associated with distributed object computing systems arises from both the dynamics of individual components and the structural relationships between components. Design decisions affecting the dynamics of individual components and in coupling and structuring these components often have significant impacts on the overall behavior and performance of the system under development or modification. For example, processor speed and memory selections impact job throughputs. Buffer size and network bandwidth drive the queuing and dropping of traffic. Choices in networking technologies constrain communication protocol options, and impact channel error rates. Communication failure and recovery schemes and mechanisms control network behavior under stress and load. Distribution of software objects across processing nodes affects processing workloads and network traffic loads. The level of multi-threading

implemented within software objects determines behavior in handling multiple, simultaneous invocation requests. Exploring these design decisions and alternatives is the focus of our interest.

The results of this research provide a modeling and simulation framework to enable exploring the dynamic behavior consequences of design decisions. This framework enables system designers to model software objects, hardware components, networking protocols, and distribution of the software objects across the networked hardware components. We call the resulting framework DEVS-DOC. DEVS refers to the underlying model specification formalism, the Discrete Event System Specification; and DOC refers to the application context, Distributed Object Computing systems. In the following, we briefly review related technologies and their relationship to DEVS-DOC.

1.2.1. Java

Java has generated a lot of excitement in the programming community with its promise of *portable* applications and applets. In fact, Java provides three distinct types of portability: source code portability, CPU architecture portability, and OS/GUI portability. These three types of portability result from the packaging of several technologies – the Java programming language, the Java virtual machine (JVM), and the class libraries associated with the language. The Java programming language provides the most familiar form of portability – source code portability. A given Java program *should* produce identical results regardless of the underlying CPU, operating system, or Java compiler. The JVM provides CPU architecture portability. Java compilers produce

object code (J-code) for the JVM and the JVM on a given CPU interprets the J-code for that CPU. To facilitate OS/GUI portability, Java provides a set of packages (*awt*, *util*, and *lang*) for an imaginary OS and GUI [Rou97].

The Java programming language embraces the object-oriented paradigm and provides a means for developing distributed software objects and applications. DEVS-DOC is intended to provide a means to model the dynamics of such distributed objects and simulate the resulting models to support design decisions on structuring and distributing the software objects being coded.

A DEVS-DOC environment has been implemented using Java technologies. As complex simulations tend to be computationally demanding, and any non-trivial DEVS-DOC model will be, improving Java performance is advantageous. Improving Java runtime performance is an active area of research [Fox96, Ins98, and Wir99].

1.2.2. Unified Modeling Language

The software systems being developed today are much more complex than the human mind can generally comprehend. To simplify this complexity, software developers often model target systems. Typically, no one model is sufficient; so, several small, nearly independent models are developed. The Unified Modeling Language (UML) [OMG99] is a graphics based language for specifying, visualizing, constructing, and documenting such software models. The development of UML has incorporated ideas from numerous methodologies, concepts, and constructs. The common syntactic

notation, semantic models, and diagrams of UML facilitate comprehension of the designed structure, behavior, and interactions of the software system under development.

UML collaboration diagrams and sequence diagrams provide a static illustration of behavior. UML deployment diagrams provide a mapping of software to hardware configurations. UML component diagrams illustrate software structures. These UML diagrams provide a graphical complement to the DEVS-DOC specification of the software components and the software distribution across hardware. Extending these UML diagrams with details on object size, message size, and object method computational workload estimates provides a complete set of parameters needed for a DEVS-DOC model and simulation.

1.2.3. Rational Unified Process

The UML is a generic modeling language that can be used to produce blueprints for a software system. To better leverage the UML, Rational Software has developed the Rational Unified Process (RUP) [Rat99] as a generic set of steps to develop such software blueprints using the UML. The idea behind the RUP is to define *who* does *what*, *when*, and *how* during the development of a software system. RUP defines a process for constructing models of a software system.

The RUP has steps for an architect to develop an architectural view of the target software system, early on, during the *analysis and design* workflow. In particular, the architect develops the software architectural design, flushes out software concurrency

issues, and determines software object distribution. The details of these steps complement development of DEVS-DOC components and couplings.

1.2.4. Co-design

Co-design is a set of engineering processes to simultaneously consider hardware and software constraints. Traditional co-design efforts are focused on enabling better communications between hardware and software developers in the design of embedded systems, such as portable devices. Our intent is to expand such capabilities into the arena of distributed co-design. We define *Distributed Co-design* as the activities to concurrently design hardware and software layers within a distributed system. Distributed Co-design can be characterized in terms of its underlying hardware and software options. Traditional Co-design assumes single or multiple software components (possibly multithreaded) being executed on a single machine having one or more processors. Distributed Co-design, however, assumes a network of distributed machines servicing parallel and/or distributed software components. Specifically, distributed hardware-layer design efforts study alternative high-level hardware topologies whereas distributed software-layer design efforts study software components characteristics and interoperability.

1.3. A Distributed Object Computing Model

The conceptual approach for this research was inspired with the Distributed Object Computing (DOC) modeling framework proposed in [But94]. This modeling framework is defined with two layers — Distributed Cooperative Objects (DCO) and

Loosely Coupled Networks (LCN) — and a mapping (Object System Mapping(OSM)) between the two layers. Part of this research involved implementing key DOC framework objects in DEVSJAVA [AIS99] and extending them to enable not only quantum simulations — the modeling of bulk behavior with random variables — but, also, directed simulations — the means to model specific underlying system technologies, structures, and behaviors.

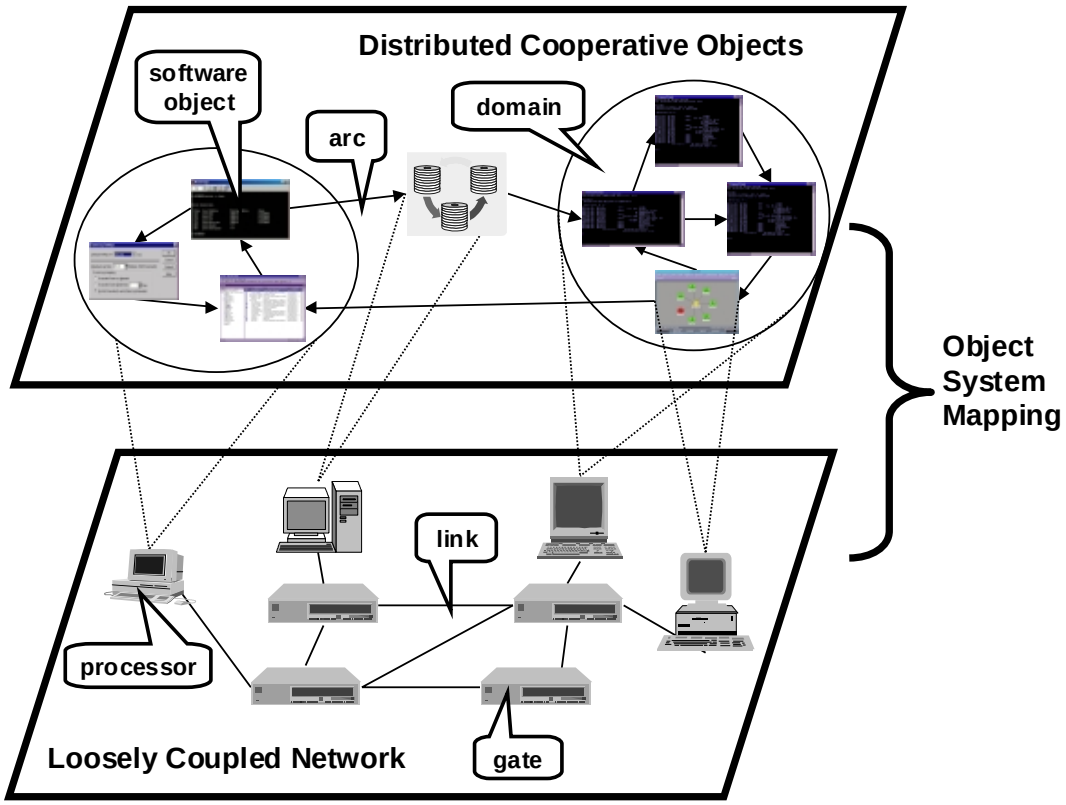


FIGURE 1. Butler's Distributed Object Computing Model

For a DOC model, Butler developed a formal set-theoretic representation of the LCN, DCO, and OSM structures. Development of behavior specification for these

structures, however, was not addressed. A central contribution of this dissertation is a formal development of such behaviors to support both quantum and directed simulations. Figure 1 depicts the LCN and DCO layers and their mappings. Appendix A provides a summary of Butler's formal set-theoretic representation.

The LCN layer provides the means to define and structure (couple) hardware components. Developing an LCN model results in the definition of the hardware architecture for a set of networked computers. The components of the LCN are processors, networking devices (hubs and routers), and network links. A processor performs computational work at a rate based on resource constraints, e.g., processor speed and memory capacity. A network device provides communication services in routing network traffic between incident network links under the constraints of buffer sizes and internal data rates. A hub device will broadcast incoming traffic out to all other outgoing links, while a router device will route traffic out to a single link towards its destination. Network links connect one or more processors and/or network gates. Network links constrain inter-processor communications based on bandwidth, channels, and error rates associated with the link.

The DCO layer provides the means to define software components and their interactions using a distributed object paradigm. Software objects within the DCO model follow a traditional object orientation metaphor; namely, they contain a set of *attributes* to define the state of the object, and a set of *methods* that operate on those attributes. The collective allocation requirement for the attributes determines the size of the software object. Each method has an associated computational workload. Software objects

interact based on computational progress. They interact via invocation and message arcs. Invocation arcs may represent either synchronous or asynchronous client-server interactions. Message arcs represent peer-to-peer message passing interactions. DCO software objects can also be organized into computational domains, which represent independent executable programs.

The OSM defines the distribution of the DCO software objects across the LCN processors. Individually, neither the LCN nor the DCO models are of any great value in modeling the behavior or performance of a distributed object computing system. The LCN provides a model of a target hardware architecture that imposes time and space constraints, but lacks a specification of dynamic behavior. Dynamic behavior is specified in the DCO model with the definition of computational loads and object interactions, which provide a model of the target software architecture. Yet, the DCO representation is independent of time and space. Mapping DCO software objects onto LCN processing nodes results in constraining the abstract dynamics of the software architecture in accordance with the time and space limitations of the hardware architecture.

1.4. A Modeling and Simulation Formalism

Using set theory, Butler provides a mathematical basis for specifying a static, structural model of a DOC system. While the intention to develop a simulation facility is clear, a target system specification formalism was not identified.

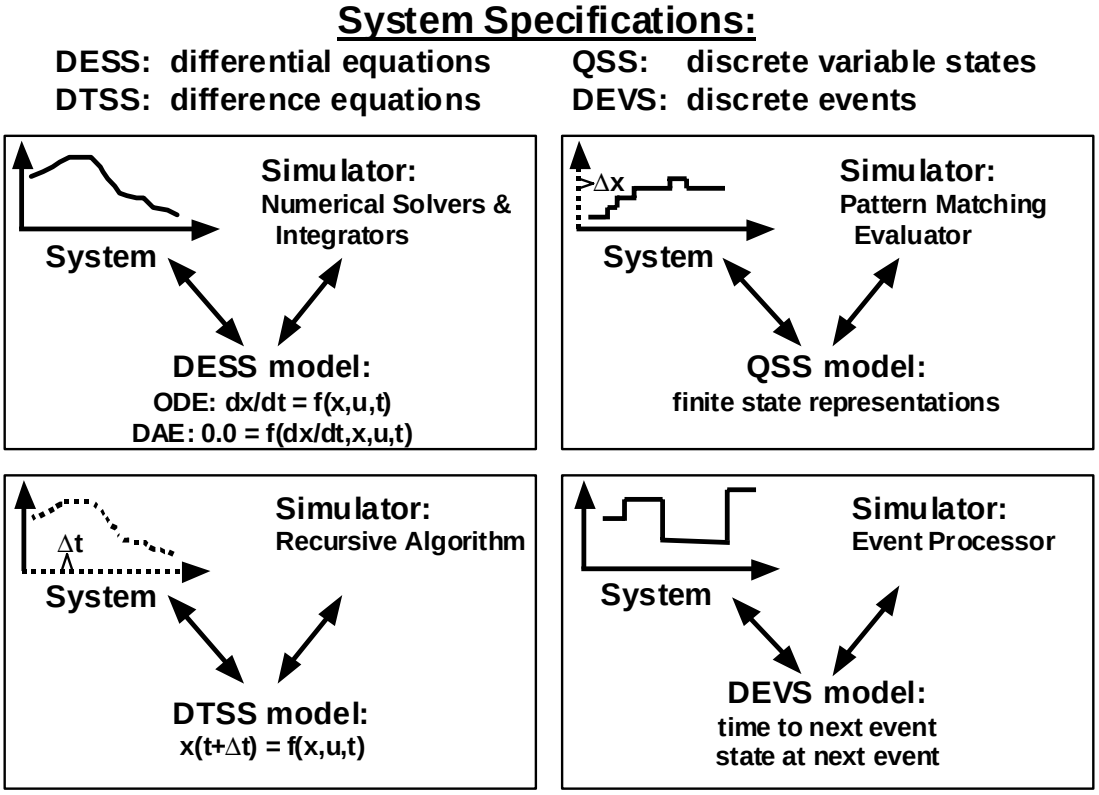


FIGURE 2. System Specification Formalisms

Four fundamental methods [Cel91, pp. 11-15] for specifying dynamical system models are Differential Equation System Specification (DESS)[1][Cel91], Discrete Time System Specification (DTSS)[2] [Cel91], Qualitative System Specification (QSS)[3] [Cel91],

[1] The *Differential Equation System Specification (DESS)* assumes that the time base is continuous and that the trajectories in the system database are piecewise continuous functions of time. The models (system specifications) are expressed in terms of differential equations (*ordinary differential equations* and/or *partial differential equations*) that specify change rates for the state variables. The corresponding simulation concept is that of numerical solvers — numerical integrators and differential algebraic equation (DAE) solvers.

[2] The *Discrete Time System Specification (DTSS)* assumes that the time base is discrete so that the trajectories in the system database are sequences anchored in time. The models are expressed in terms of difference equations that specify how states transition from one step to the next. A forward marching time-stepping algorithm constitutes the associated simulator.

and Discrete Event System Specification (DEVS)[4].[Zei76, Zei84, Zei91, and Zei99c]. Figure 2 depicts the basics of these four fundamental specification methods. In order to develop a dynamic system specification for DOC models, we needed to select one of these fundamental approaches. As is detailed later, the behavior of interest in DOC components is piecewise constant over variable periods of time. Thus, the DEVS formalism provides a straightforward means to specifying the necessary DOC component behaviors.

1.5. The DEVS-DOC Framework

The remainder of this dissertation introduces the DEVS-DOC modeling and simulation environment. This environment enables system designers and developers to independently model the hardware and software architectures and then map software to hardware to form a distributed object computing model ready for simulation. The model is easily wrapped within an experimental frame — instrumentation to collect simulation results, statistics, and trajectories — to aid evaluation and analysis of the system behavior.

The capability to independently model the hardware and software architectures facilitates *concurrent engineering* practices, as well as introduces the concept of

---

[3] The *Qualitative System Specification (QSS)* assumes a continuous time base and a finite set of values for state variables. The models are a posteriori derived patterns of system states. Simulators evaluate inputs against these patterns to determine system outputs and responses.

[4] The *Discrete Event System Specification (DEVS)* assumes that the time base is continuous and that the trajectories in the system database are piecewise constant, i.e., the state variables remain constant for variable periods of time. The jump-like state changes are called *events*. The models specify how events are scheduled and what state transitions they cause. Associated simulators handle the processing of events as dictated by the models.

distributed co-design. This approach provides a means to bridging the gap between hardware, software, and systems engineering, and enables system designers and developers to focus on the complexity issues associated with the interdependencies and inter-dynamics of processor behaviors, communication protocols, networking topologies, and software structures.

1.6. Summary of Contributions

Overall, the primary contribution of this thesis is in the demonstration of a practical methodology for modeling and simulating the complex dynamics and interactions of distributed computing systems. Additionally, this thesis demonstrates the suitability of DEVS in representing and simulating DOC models; reveals how maintaining a separation of concerns contributes to structural modeling; introduces the concept of Distributed Co-Design; advances the experimental frame concept with layers; and shows the expediency and simplicity of aggregated couplings.

This research demonstrates a practical methodology to modeling and simulation DOC systems with a concrete realization of Butler's conceptual model. Using formal methods (DEVS), the realization involved identifying and developing behavioral representations of the critical dynamics associated with hardware and software components. Conversely, the representation of DOC models in DEVS demonstrates the applicability of DEVS in the distributed computing problem domain.

A key aspect within our DEVS-DOC realization is a separation of concerns; in particular, maintaining independence of the software (DCO) and hardware (LCN) models

and only introducing interdependencies in the distribution (OSM) of software across hardware. Under this approach, the software behavior is modeled independent of time and space constraints, whereas the networked hardware components impose such constraints. The networked hardware imposes time constraints in the processing of software object jobs and the communication of software object interactions. The networked hardware imposes space constraints based on the mapping of software objects onto the hardware and the networked topology of the hardware components. This level of independence between the DOC components (LCN, DCO, and OSM) facilitates structural approaches to implementing the models.

The concept of Distributed Co-Design expands on the idea of concurrent engineering of hardware and software constraints with the added dimension of networked systems. Through the DEVS-DOC environment, we advance an approach of independently modeling the hardware and software systems. The hardware representations explicitly support investigations into the implications of distributed processors, networked components, communication protocols, and network topologies. The software representations explicitly support investigations into multi-threading, invocation queuing, parallel processing, and software object interactions. The coupling of the hardware and software representations enables investigations into computational load balancing across processors, message traffic loading across network components, and execution latency as a combination of computational loading, job queuing, and communications latency.

The experiment frame concept is advanced with the idea of layered frames. In particular, in the case studies of this thesis, a layered experimental frame is utilized to investigate model sensitivities.

In realizing the DEVS-DOC environment, object-oriented techniques are applied to facilitate a modular and hierarchical system construction. With this object-orientation, DOC specific "aggregated coupling" mechanisms are developed to significantly simplify the specification of the LCN topology, the OSM specification of software to hardware mappings, and the specification of the experimental frame couplings.

With the DEVS-DOC environment and case studies, we shall demonstrate systems analysis of DOC implementation technologies, DOC designs against “inherent” and “accidental” complexities, software multithreading behavior and performance, network topology and protocol selections, and software partitioning across hardware.

1.7. Plan of Dissertation

The next chapter of this dissertation surveys literature related to the research presented in subsequent chapters. In chapter 3, the conceptual constructs and formalisms used to develop the DEVS-DOC environment are outlined. Chapter 4 introduces the DEVS-DOC modeling and simulation environment. This chapter provides details on the structural representation of the DEVS-DOC modeling components and experimental frame. Chapter 5 complements the structural details of chapter 4 with details on the formal specification of behavior dynamics for the DEVS-DOC components. Then, in chapter 6 we present four distributed system case studies using the DEVS-DOC modeling

environment. The first case study models Simple Network Management Protocol (SNMP) interactions between a network management station and a set of networked devices; the second study models the interactions of an HLA-compliant distributed simulation federation [Zei99c]; the third models the interactions of an email application [Hil98b]; and the fourth case study looks at LCN alternatives for the email application. Chapter 7 concludes the dissertation with a summary of contributions and a discussion of future work.

2. RELATED WORK

This chapter presents an overview of work related to this research. We organize this synopsis of related work along the lines of our DOC abstractions. Related to the LCN abstraction is work in the area of networked information technology modeling: communications, protocol, network, and processor modeling; related to the DCO abstraction is work in the area of software systems modeling; and the most closely related work to the OSM abstraction is research in the area of Co-design. Finally, we relate this modeling and simulation research to alternate distributed system design and analysis approaches.

2.1. Networked Information Technology Modeling

2.1.1. Queuing System Approaches

A key performance metric of an information system is average delay or waiting time. Queuing theory is a formal methodology for analyzing network delays based on statistical predictions. Queuing theory has its roots in the early twentieth century studies of the Danish mathematician A. K. Erlang on telephone networks and in the creation of stochastic models by the Russian mathematician A. A. Markov. The simplest queuing model consists of a single queue and server. Using Little's Theorem, $N = \lambda / \mu$, we can analyze a variety of situations based on the mean arrival rate λ, the mean service rate μ, and the mean number of customers or objects being served by the system N. Queuing theory continues to evolve, and enables analytical analysis of more complex queuing

systems, networks of queues and servers, and various statistical behaviors associated with arrival rates and service rates.

The approach taken in this research, however, has attempted to avoid some of the simplifying assumptions required when taking a queuing theory approach. One objective is to enable modelers with a means to construct arbitrary structures of software objects, network components and topologies, and their subsequent coupling. Arbitrary networks of *queuing systems* tend to be problematic and can often violate assumptions about the random distributions of arrivals.

2.1.2. Discrete Event Approaches

Our discrete event systems approach to modeling and simulating communications networks and processors is not unique. A variety of commercial and academic efforts have been invested in developing modeling and simulation capabilities to support analysis of network components and technologies, routing strategies, protocols, network controls and recovery mechanisms, traffic flows and loads. OPNET™ Modeler [MIL99] and COMNET III™ [CAC99] are two examples of commercially available tools for discrete event modeling and simulation of communications networks, devices, and protocols. In these tools, modeling of the information flows and loads to stimulate the system under investigation is accomplished with setting traffic parameters associated with selected nodes in the model. With DEVS-DOC, we take a more direct means of modeling distributed software systems and then support investigating alternative mappings of the software across the networked components.

2.2. Software Systems Modeling

2.2.1. Object-Oriented Design

Several prominent object-oriented design notations have recently been combined to form the Unified Modeling Language (UML) [OMG99]. The UML provides a family of graphical notations for describing the attributes and relations between objects under design. These graphical notations support the production of static structure diagrams and behavior diagrams. The structure diagrams include *class* and *object* diagrams, while the behavior diagrams include *use-case*, *interaction*, and *activity* diagrams. The structure diagrams identify definitional and referential relationships among components. The definitional relationships create taxonomic hierarchies, and the referential relationships create compositional hierarchies.

While the structure diagrams do not describe the interaction and behavior of components, UML behavior diagrams can provide a static view. UML sequence and collaboration charts are interaction diagrams that depict the flow of events, messages, and actions among interacting software components. UML state charts depict component behavior in terms of state changes and transitions. However, these diagrams are independent of each other and do not systematically relate patterns of interaction and state changes and transitions, nor do these charts relate resource dependencies, workloads, or relative timing implications. Thus, reasoning about, and evaluating, software component interaction performance and behavior is difficult. Within the DEVS-DOC environment, we couple the software component workloads and interactions to the

LCN, which imposes resource constraints, enabling the evaluation of these interactions and the resulting resource utilization impacts.

2.2.2. Software Architecture Modeling

To deal with the increasing complexity of software systems, the overall system structure — or software architecture — is gaining increasing attention as a central design problem. WRIGHT [All97] is a software architectural description language that provides a formal means to describe architectural configurations and architectural styles. The software architectures are defined in terms of software components and connectors (interaction patterns). The descriptions of software components and connectors are "architectural" in that the notation defines behavior independent of implementation or formal programming language constructs. The practicality of a formal architectural description language is in the ability to analyze component interactions and patterns of interactions for semantic correctness.

2.3. Co-design

Research in co-design is closest to our work when viewed from an abstract view of concurrent hardware and software analysis and design within a framework. Given research activities in Co-design since the 1980s, many of the developed methodologies are primarily focused on real-time embedded systems such as portable devices [Yen97, Roz94, Sch98, and Fli99]. These Co-design methodologies and their supporting environments primarily emphasize concurrent engineering of hardware and software constraints (e.g., size, timing, weight, cost, reliability, etc.) for a given a piece of

hardware. While Co-design and the work described in this dissertation are both focused on concurrent hardware and software engineering issues, this dissertation is distinguished by its focus on analysis and design of distributed systems; systems where networked hardware components enable *physically distributed cooperating* software components to interact. DEVS-DOC expands the basic ideas of Co-design system development from the execution of one or more processes on a single device (embedded system) to the inter-dependent execution of many processes running on multiple, distributed and networked, heterogeneous devices (system of systems).

Examples of research activities in Co-design include CASTLE at GMD Germany [GMD99], Model-based Co-design at the University of Arizona [EDL99], and POLIS at the University of California, Berkeley [Chi94]. CASTLE is a set of tools to support synthesis of embedded systems. CASTLE's tool set can be used to convert system's software and hardware specifications via System Intermediate Representation using input and output processors from one to the other. For example, CASTLE's co-synthesis environment can generate a processor's VHDL description, and a compiler can translate any C/C++ program from a given application domain into the operational code of the intended processor. Similarly, POLIS provides a Co-design Finite State Machine representation that can be used to characterize both hardware and software components of a system using tools such as VHDL or graphical FSMs. The environment supports formal verification, simulation, and HW/SW system level partitioning as well as HW and SW synthesis and their interfacing. Model-based Co-design (and its computer-based design environment, SONORA [Roz99]) provides a methodology that is comprised of

five stages: (1) specification (requirements and constraints), (2) modeling, (3) simulation/verification, (4) model mapping to hardware/software components, (5) prototyping and implementation. Given CASTLE, POLIS, and Model-based Design, from a framework approach, the latter is the closest to our work. However, while this framework and others promote system specification into hardware/software architectures, they do not support concurrent hardware/software analysis and design within a distributed context. DEVS-DOC does support this distributed context.

2.4. Alternative Analysis and Design Approaches

2.4.1. Distributed Simulation Exercise Construction Toolset (DiSect)

An environment known as DiSect has been proposed to aid the development, execution monitoring, and post execution analysis of a distributed simulation [STR99]. DiSect is comprised of three software tools. The Exercise Generation (ExGen) tool aids developers in composing simulations from a web-based simulation object repository. The Distributed Exercise Manager (DEM) is the simulation execution monitoring tool, which controls simulations and monitors the distributed simulation infrastructure: workstations loads, network loads, and the High Level Architecture (HLA) Run Time Infrastructure (RTI) [DMS99]. The Modular After Action Review System (MAARS) provides tools for visualization and analysis of data collected during a DEM simulation. DiSect was implemented to support evaluation and analysis of Army training simulation exercises during, and after, the execution of the simulation exercise.

While such an environment can aid in the re-design of systems of interest, this approach is limited to a narrow scope of distributed systems — Army training simulation exercises in an HLA/RTI environment — and to post-mortem system evaluation. System analysis is based on the monitoring of the operational system. The DEVS-DOC approach provides a modeling and simulation environment enabling system evaluation and analysis prior to composing and running the real target system.

2.4.2. Petri Nets

Petri Nets provide an abstract state-machine model of a system. Petri Net models allow analysis of system concurrency, asynchronous operations, deadlocks, conflicts, and event-driven actions. In short, they offer a means to develop and evaluate the logical structure of a system. Petri Nets are composed as directed graphs consisting of two classes of nodes, called *places* and *transitions*, that are interconnected with *arcs*. The state, or *marking*, of a Petri Net is defined with the placement of *tokens* on *places*. Allowed state changes are defined with the association of *firing rules* to *transitions*. Variations on the Petri Net methodology can add attributes to *tokens* — called *colored* Petri Nets — as well as timing, conditional, and probabilistic firing details to *arcs*.

Petri Nets provide a means to define the set of possible execution traces through a system. Each trace represents a possible path through the state graph. This modeling approach is often used to ensure that there is always a transition possible for any input, and that selected, undesired states are not reachable.

With only two types of nodes, *places* and *transitions*, the Petri Net modeling notation makes no distinction between hardware and software components. Due to this semantic limitation, Petri Nets provide no direct means to model a distributed object computing system. Use of the Petri Net formalism as the basis for building higher level abstractions of DOC hardware and software components is a potential area of research.

3. DEVS-DOC BUILDING BLOCKS

In developing an environment for modeling and simulating distributed object computing systems, we utilize the DEVS formalism to develop specifications of DOC components; implement these representations using DEVSJAVA; and, utilize the experimental frame concept to define simulation experiments. As already mentioned, the DEVS formalism provides a systems theoretic basis for specifying component behaviors. Implementing in DEVSJAVA enables object-oriented modeling, concurrent simulations, concurrency among interacting simulation objects, and web-enabled simulations. The experimental frame concept provides a formal structure when specifying the simulation conditions to be observed for analysis. These conceptual constructs are outlined in this chapter as background for the following two chapters that detail the DEVS-DOC modeling and simulation environment developed during this research. We begin this chapter with an overview of Butler's formal structure for modeling a DOC system.

3.1. A Distributed Object Computing Model

Butler developed a formal representation for modeling DOC systems. This formalism allows for independent development of representations for the hardware and software architectures, and then coupling these models into a systems representation. The hardware architecture, called the Loosely Coupled Network (LCN) model, defines the real-world network topology interconnecting computing systems. The software architecture, called the Distributed Cooperative Object (DCO) model, defines object-oriented software components and structures. The DCO software objects are mapped

onto LCN hardware components to couple the two models to form an Object System Mapping (OSM). These components are assigned random variables (e.g., bandwidth, error rates, compute loads, data sizes, etc.) to provide the basis for simulating dynamic behavior. A summary of Butler's formal notation is provided in Appendix A.

3.1.1. Hardware: Loosely Coupled Networks

The LCN representation of networked computer components results in the specification of processors, network gates, and links. The processors serve as nodes, on which software objects of the DCO abstraction may be loaded and executed. The two critical parameters for these processing nodes are processor storage size and processor speed. The storage size of the processor constrains software objects in their competition for memory resources. The processor speed constrains the rate at which software jobs are processed. The network gates in the LCN represent hubs and routers in a computer network. The critical parameters constraining the performance of the gates are operating mode (hub or router), buffer size, and bandwidth. The links model the communication media between processors and gates. Critical parameters for links include number of channels, bandwidth per channel, and error rates. The LCN network topology is defined by mapping the processors and gates onto the links.

Figure 3 depicts an example LCN structure of two computing locations interconnected with a T1 carrier link, L_0. The left-side local area network (LAN) has a gate, G_0, configured as a router to interconnect the two ethernet links, L_1 and L_2, and the T1 carrier link. The right-side FDDI star LAN has a gate, G_1, configured as a router, to

interconnect the T1 carrier with the FDDI star LAN. Gate G_2 is configured as a hub to create the FDDI LAN star topology.

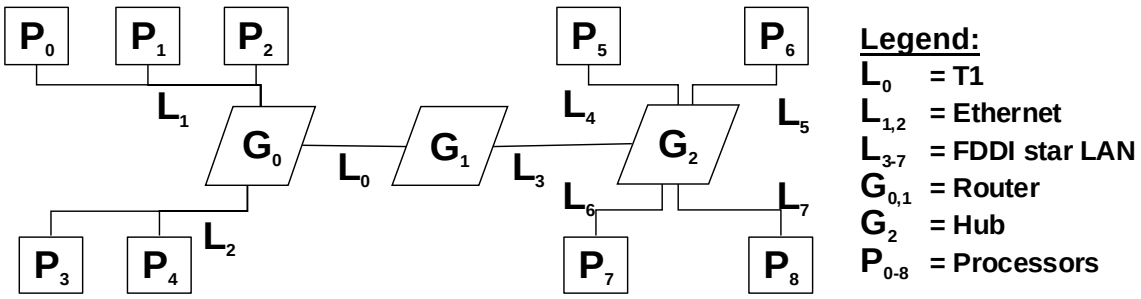


FIGURE 3. Example LCN Structure

3.1.2. Software: Distributed Cooperative Objects

The DCO abstraction of software components results in the specification of computational domains, software objects, methods, and object interaction arcs. A set of software objects forms a computational domain. Any software object may belong to more than one computational domain. A computational domain represents an executable environment or program and identifies *initializer* software objects. These *initializer* software objects stimulate the initial execution order of software objects in a domain.

A software object is based on the object-oriented concept of an object that contains both *attributes* (data members) and *methods* (functions) that operate on the attributes. The collective memory requirements of these attributes and methods characterize the object's size. When the software object is invoked, the size parameter loads the supporting LCN processor memory. The object may be invoked autonomously or on the receipt of an object interaction arc. The software object has a thread mode parameter that defines the granularity of its multi-threaded behavior: *method*, *object*, or

none. This thread mode determines the level of execution concurrency the software object supports. At the method level, all requests to the object may execute concurrently. At the *object* level, only one request per method may execute concurrently; additional requests against an executing method get queued. At the *none* level, only one request per object may execute at a given time; each additional request to the object gets queued.

The *methods* of a software object are characterized as a computational work load factor and an invocation probability. The work load factor represents the amount of computational work required to fully execute the method. The invocation probability is an artifact of the quantum modeling technique. From the *quantum* perspective, when a software object is invoked, it is irrelevant which method is actually selected as long as all the methods of the object are invoked in correct proportion.

Two types of software object interactions are identified, *message arcs* and *invocation arcs*. A *message arc* represents peer-to-peer exchanges between objects. When a source object sends a message, it may target several destination objects. The frequency of firing a message arc is based on the computational progress of the source software object. The message size parameter characterizes the amount of data transmitted. An *invocation arc* represents client-server type interactions between two software objects. When a client object fires an invocation arc, the message size parameter specifies the amount of data sent. The destination server object invokes a method and, on method completion, sends a response back to the client. Invocation arcs have either a *synchronous* or *asynchronous* blocking mode. In synchronous mode, the

firing method is blocked from further execution until the return message is received. In asynchronous mode, the firing method is allowed to continue execution.

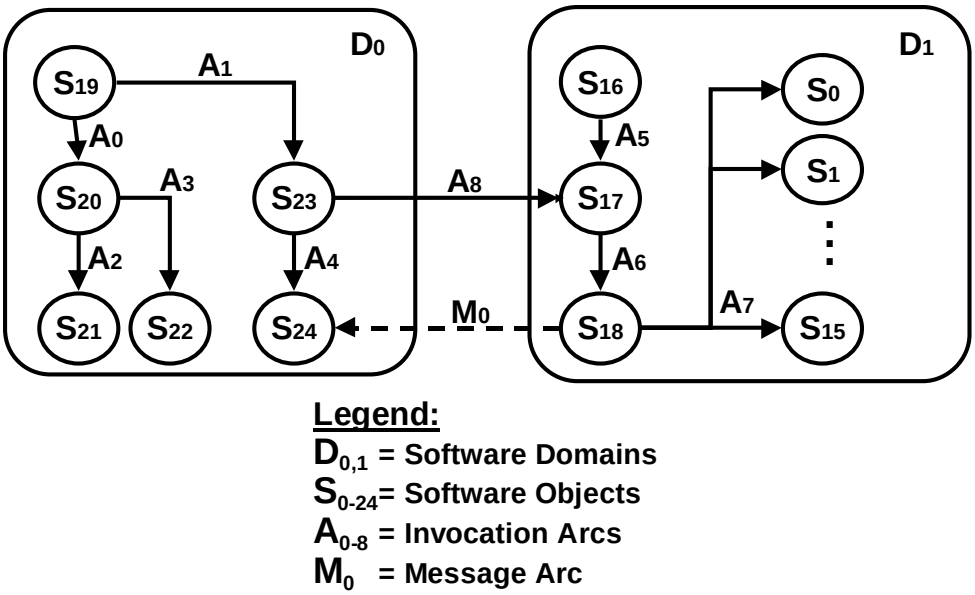


FIGURE 4. Example DCO Software Structure

Figure 4 depicts an example DCO abstraction of 25 software objects organized into two computational domains. Domain D_1 acts as a compute server with 16 parallel compute objects. Client S_{23} from D_0 requests service with invocation arc A_8 and invokes S_{24} to wait for the results via message arc M_0. When S_{17} receives a request from the client, it invokes S_{18}, and S_{18} distributes the required computation over $S_{0\text{-}15}$. When all 16 compute objects complete and return results to S_{18}, the results are sent back to the client domain to S_{24}.

Software object methods and arc interactions stimulate the dynamic behavior of distributed object computing systems. DCO software objects interact with other objects via invocation and message arcs. In cases where software objects are executed on

different processors, the invocation and message arcs are routed as network data traffic through the LCN processors, gates, and links. When invoked by an incoming arc, a software object is loaded into processor memory. Arc reception also triggers selection of a method for execution. The selected method is either fired for execution or queued based on the multi-threading mode and the current state of the software object. Fired jobs go to the OSM-assigned processor for execution. Completed jobs are returned from the processor indicating *computational progress* for the software object. Based on computational progress, additional interaction arcs are selected for exchanges with other DCO software objects. Selected arcs are fired / transmitted across the LCN. Methods that trigger arc firings continue execution unless blocked by the firing of a *synchronous* arc. In this case, the method continues execution after the associated return for the synchronous arc is received. When all methods complete execution, and when all fired arcs expecting return arcs are received; the object unloads processor memory and inactivates.

3.1.3. Distribution: Object System Mapping

Individually, neither the LCN nor the DCO models are of any great value in modeling the behavior or performance of a distributed object computing system. The LCN provides a model of the target hardware architecture that imposes time and space constraints. The LCN, however, lacks a specification of dynamic behavior. Dynamic behavior is specified in the DCO model with the definition of *initializer* objects, object methods and arc interactions. These DCO definitions provide a model of the target

software architecture. However, this DCO behavior representation is independent of time and space. By mapping the DCO software objects onto LCN processing nodes, the abstract behavior dynamics of the software architecture are coupled with and constrained by the capacity of resources (processor speed, memory, bandwidth, buffer size, etc.) and topology of the hardware architecture. The DCO onto LCN mapping is referred to as the Object System Mapping (OSM).

The performance and behavior of the resulting distributed object computing system can support design analysis with simulation. During simulation, the invocation of a software object competes for processor resources in terms of storage space to load the object and processor speed to execute its methods. These dynamics drive the performance of the processor as it serves OSM-assigned software objects. Processor performance also impacts performance of OSM-assigned software objects in terms of completing computational tasks (object methods) and speed of network data (DCO arc) exchanges between software objects. These data exchanges also drive the dynamics of the network performance as well as the performance of the DCO domain (application) as software object exchanges denote computational progress between objects.

In addition to mapping DCO software objects onto LCN processors, the OSM model defines a set of communication modes and maps the DCO interaction arcs into these communication modes. The communication modes specify how DCO interaction arcs are processed into packets that will transit the LCN. The communication mode defines constraints on packet size, packet overhead size, and packet acknowledgment size.

3.2. Discrete Event System Specification

The DEVS modeling approach supports capturing a system's structure from both functional and physical points of view. A DEVS model can be either an *atomic* model or a *coupled* model [Zei84]. The *atomic* class realizes the basic level of the DEVS formalism with specifications of the elemental components of the system of interest. The *coupled* class realizes the compositional constructs for structuring DEVS models into system hierarchies. Atomic and coupled models can be simulated using sequential computation or various forms of parallelism [Cho96].

A DEVS atomic model specification defines the states (variable values) and associated time bases resulting in piecewise constant trajectories over variable periods of time (see footnote 1). The atomic model specification also defines how to generate new state values and when new states should take effect. A Parallel DEVS *atomic* model [Cho96] is formally defined as:

$M = < X_M, Y_M, S, \delta_{int}, \delta_{ext}, \delta_{conf}, \lambda, ta >$
where
$X_M = \{(p,v) \mid p \in IPorts, v \in Xp\}$ is a set of input ports and values
$Y_M = \{(p,v) \mid p \in OPorts, v \in Xp\}$ is a set of output ports and values
S is a set of states
δ_{int}: $S \rightarrow S$ is the *internal transition* function
δ_{ext}: $Q \times X_M^b \rightarrow S$ is the *external transition* function
δ_{conf}: $Q \times X_M^b \rightarrow S$ is the *confluent transition* function
λ: $S \rightarrow Y_M^b$ is the *output* function
ta: $S \rightarrow \Re^+_{0,\infty}$ is the *time advance* function.
with
$Q = \{(s,e) \mid s \in S, 0 \leq e \leq ta(s)\}$ is the set of *total states*
e is the *time elapsed* since last transition
X_M^b is a bag of inputs (a set with possible multiple occurrences)
Y_M^b is a bag of outputs (a set with possible multiple occurrences)

A DEVS coupled model designates how (less complex) systems can be coupled together and how they interact with each other. Given atomic models, DEVS coupled models are formed in a straightforward manner. Two major activities involved in *coupled* models are specifying its component models, and defining the coupling that represents desired interactions. A Parallel DEVS *coupled* model [Cho96] is formalized as:

$DN = < X, Y, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\} >$
Where
X is a set of input values
Y is a set of output values
D is a set of the DEVS component names.
For each $i \in D$,
M_i is a DEVS component model
I_i is the set of influencees for I.
For each $j \in I_i$,
$Z_{i,j}$ is the i-to-j output translation function.

The sequential and parallel views play a central role in modeling and simulation of coupled models since each coupled model is essentially comprised of multiple atomic models. Two different formalisms have been introduced. The sequential formalism [Zei84] treats components' simultaneous transitions (δ_{ext} and δ_{int}) sequentially, while the more recent Parallel DEVS formulation [Cho96] treats them concurrently. Parallel DEVS supports (1) processing of multiple input events and (2) local control on the handling of simultaneous internal and external events.

3.2.1. DEVSJAVA

Using the Java programming language, the basic DEVS constructs have been implemented in an environment called DEVSJAVA [AIS99 and Sar97]. This

environment provides the foundation upon which higher constructs of DEVS (e.g., endomorphism and variable structure) [Zei90] can be created using the basic, as well as the internet-based features, of the Java environment. The DEVS atomic and coupled models can be developed and simulated as standalone applications or as applets. Since DEVSJAVA is a multi-threaded implementation, each simulation model is assigned a unique thread allowing for simultaneous execution of several models. Similarly, in DEVSJAVA, each atomic model can be assigned a unique thread allowing for simultaneous execution of the DEVS atomic components. These multi-threading features enable handling multiple events concurrently within the component models of a distributed object computing system.

In developing the DEVS-DOC environment, we use the DEVSJAVA implementation along with the experimental frame concept described later. Implementation of DEVS-DOC on top of DEVSJAVA enables object-oriented modeling, concurrent simulations, concurrency among interacting simulation objects, and web-enabled simulations.

3.2.2. DEVS Object-Orientation

Implementing DEVS in an object-oriented computational form leads to a natural and easy to comprehend modeling framework. A generic DEVS class hierarchy is given in Figure 5. The parental class is *entity*, from which *devs* is derived. The *devs class* is specialized into two sub-classes: *atomic* and *coupled*. Each of these classes enables system models to be expressed within the DEVS formalism outlined above.

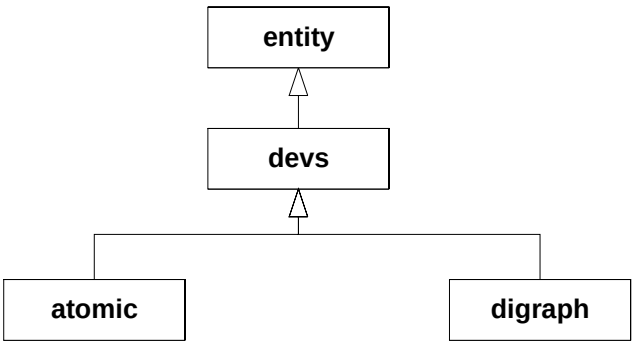


FIGURE 5. Generic DEVS Object-Oriented Hierarchy

A second class hierarchy is the heterogeneous container class library (HCCL) [Zei97a], as depicted in Figure 6, which enables easier manipulation of sets of entities. Again, the parental class is *entity*. The *container* class is a generalized form of a linked list that is based on set theory, and provides methods to store, retrieve, and organize objects. The *container* class is to define objects that contain other objects. To facilitate modeling the exchange of events between DEVS *atomic* and *coupled* models, the *message* class is used to structure the event information as output from source models and as input to destination models. The *content* of a *message* can be any instance of an *entity* or *entity* derived class.

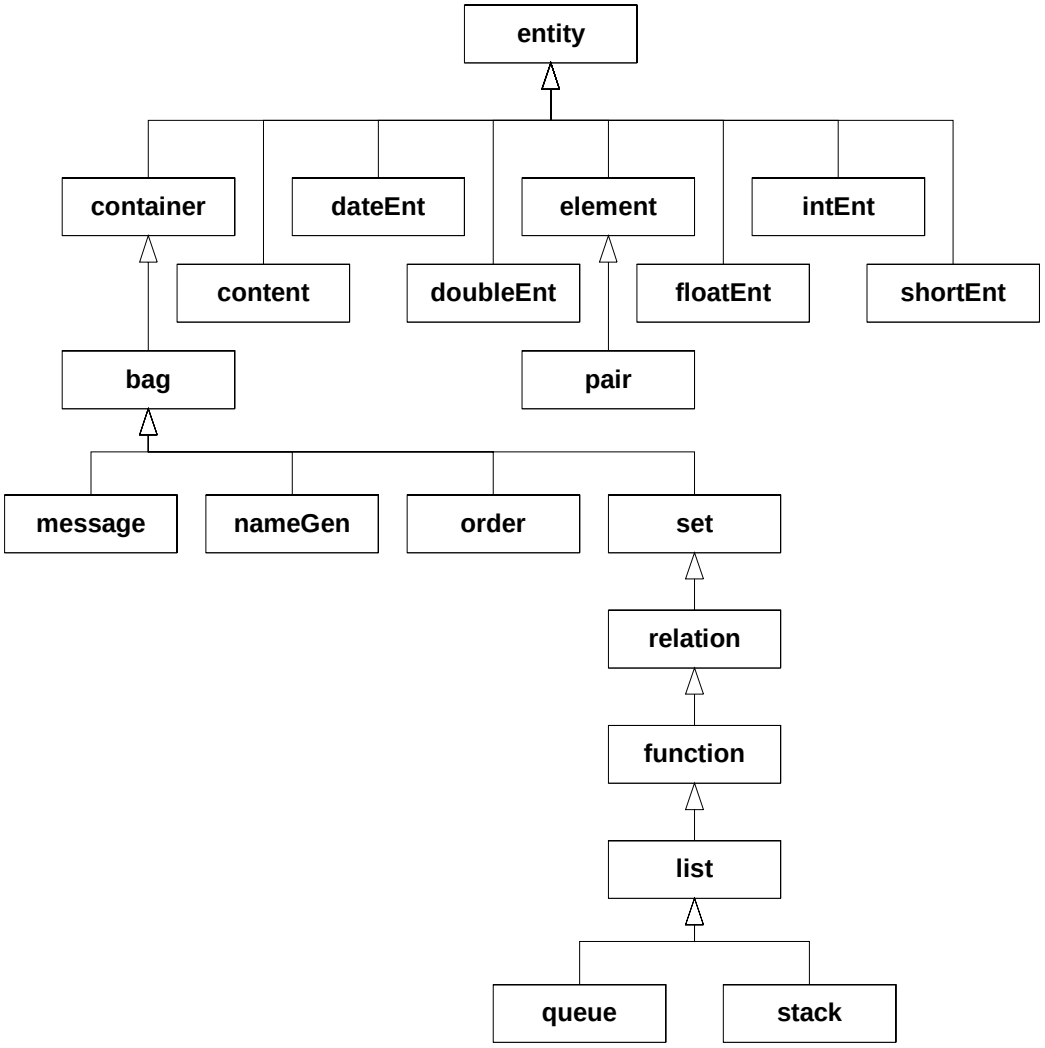


FIGURE 6. Heterogeneous Container Class Library

3.3. Experimental Frame

An experimental frame is an artifact of a modeling and simulation enterprise [Zei84]. It is a specification of the conditions under which a system is observed and experimented with. The experimental frame is the operational formulation of the objectives motivating a modeling and simulation project. A typical experimental frame,

as depicted in Figure 7, consists of a *generator*, an *acceptor*, and a *transducer*. The *generator* stimulates the system under investigation in a known, desired fashion. The *acceptor* monitors an experiment to see that desired conditions are met. The *transducer* observes and analyzes the system outputs. The experimental frame concept provides a structure to specifying the simulation conditions to be observed for analysis. DEVS-DOC specific experimental frame components are discussed later in this disseration.

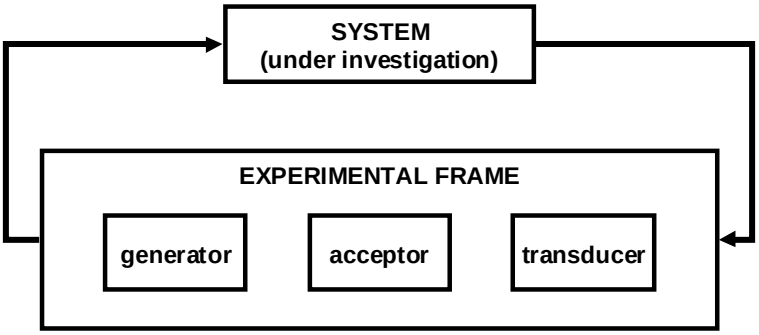


FIGURE 7. Experimental Frame

3.4. System Entity Structure

The System Entity Structure (SES) is a means of formally organizing a family of possible configurations for a system under investigation [Zei90]. This formal organization of possible system configurations bounds the system design space as a set of design alternatives. The SES formalism provides an operational language for specifying hierarchical structures. The hierarchical structures are a declarative knowledge representation of *decomposition*, *taxonomic*, and *coupling* relationships among *entities* forming the system. The *decomposition* scheme allows representing the construction of a

system from a set of *entities*. The taxonomy is a representation of possible variants of an *entity*; i.e., how the entity is categorized or sub-classified. The *coupling* knowledge identifies constraints on ways in which *entities* may be coupled together in compositions.

The SES knowledge of a system is often depicted graphically as a labeled tree. Within the SES tree, an *entity* represents a real world object that is a component of the system in one or more decompositions. An *aspect* represents how an *entity* may be broken down into sub-entities. A *specialization* is a means of classifying an entity; it expresses alternative choices for components in the system being modeled.

The nodes in an SES graph may also have attached variables. Attached variables are a means for associating attribute knowledge with the nodes. The following six axioms formally characterize the SES framework.

Uniformity:	Any two nodes that have the same labels have identical attached variable types and isomorphic sub-trees.
Strict hierarchy:	No label appears more than once down any path of the tree.
Alternating mode:	Each node has a mode that is either *entity*, *aspect*, or *specialization*; if the node mode is *entity* then the modes of its successors are *aspect* or *specialization*; if the node mode is *aspect* or *specialization* then the modes of its children are *entity*. The mode of the root is *entity*.
Valid brothers:	No two brothers have the same label.
Attached variable:	No two variable types attached to the same node have the same name.
Inheritance:	Every entity in a *specialization* inherits all the attached variables, aspects, and specializations from the parent of the specialization.

For example, in an SES for a wristwatch, the entity wristwatch may have an aspect that decomposes into a *wristband* entity and a *clock* entity. Furthermore, the *clock* entity may have a specialization that classifies the *clock* as either an *analog* or *digital* clock. A *style* attribute may also be associated with the wristwatch to specify its style as

a *men's* or *lady's* watch. The wristband may also have an attribute to specify its color. This simple SES example for a wristwatch is depicted in Figure 8.

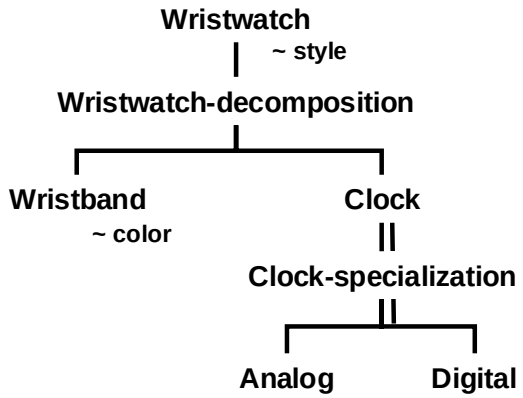


FIGURE 8. Wristwatch System Entity Structure

The entity-aspect relationship represents decomposition knowledge. The entity-specialization relationship represents taxonomic knowledge. As coupling knowledge defines how entities (models) communicate with each other, this knowledge is associated with the respective SES aspects.

The SES is a powerful mechanism for identifying the components and structure of model bases. We use the SES in section 6 as a means to depict the design space for each of the case studies.

4. DEVS-DOC MODELING AND SIMULATION ENVIRONMENT

This research has focused on developing an environment to model and simulate distributed object computing systems in support of system design and design evaluation. In this section, we describe the DEVSJAVA implementation of the DCO, LCN, OSM, and the associated experimental frame components. The implementation discussion of this section focuses on the structural representation and highlights deviations taken from Butler's approach. Throughout this section, qualitative behavior descriptions are provided to assist in defining, and rationalizing the need for the various structural elements and attributes. In the next section, we provide rigor to the specification of the behavioral representations.

The DEVS-DOC implementation has been successful in leveraging the object-oriented paradigm to maintain a separation of concerns and to encapsulate abstractions of behaviors and data. A primary separation of concerns is between the LCN and DCO abstractions. The resulting LCN model describes the networked computing hardware architecture under investigation, while the resulting DCO model describes the distributed object software architecture. The OSM abstraction describes the "mapping" of the DCO software objects onto LCN processor nodes to form a dynamic systems model.

Within the DEVS formalism, atomic models interact via DEVS messages. For the DEVS-DOC implementation, DEVS messages between DCO software objects and LCN processors contain one of two object class types: *job* or *msg*. A DCO software object sends (a DEVS message containing) a *job* to an LCN processor to signify the

execution of a method and the LCN processor returns the *job* to the DCO software object to signify execution completion. Similarly, a DCO software object sends (a DEVS message containing) a *msg* to an LCN processor to interact with other DCO software objects and the LCN processor routes the *msg* to LCN links towards the destination DCO software object.

In this chapter, we provide details on the implementation of a DOC modeling and simulation environment in DEVS. In several areas, the implementation deviates from Butler's formal structure to simplify the implementation effort, simplify the user's (a modeler's) effort, or both. In other areas, we extend the structure to enable *quantum* and *directed* modeling. These deviations and extensions are detailed within each section.

4.1. A Process For DOC Modeling

To bring a level of tractability into the design and analysis of a distributed object computing system, we decompose the modeling effort into five steps:

1) state the modeling and simulation objectives;
2) define the processing nodes and interconnection network;
3) define the software objects and their interactions;
4) associate software objects with processing nodes; and,
5) configure simulation control components.

The first step determines the target system performance and behavior questions of interest. Following the terminology and conventions of Butler, the next three steps result

in the specification of a Loosely Coupled Network (LCN), a set of Distributed Cooperative Objects (DCO), and an Object System Mapping (OSM). The fifth modeling step involves defining an experiment to stimulate the model during simulation and to collect data needed for performance and behavior analysis. This fifth step produces an Experimental Frame [Zei84].

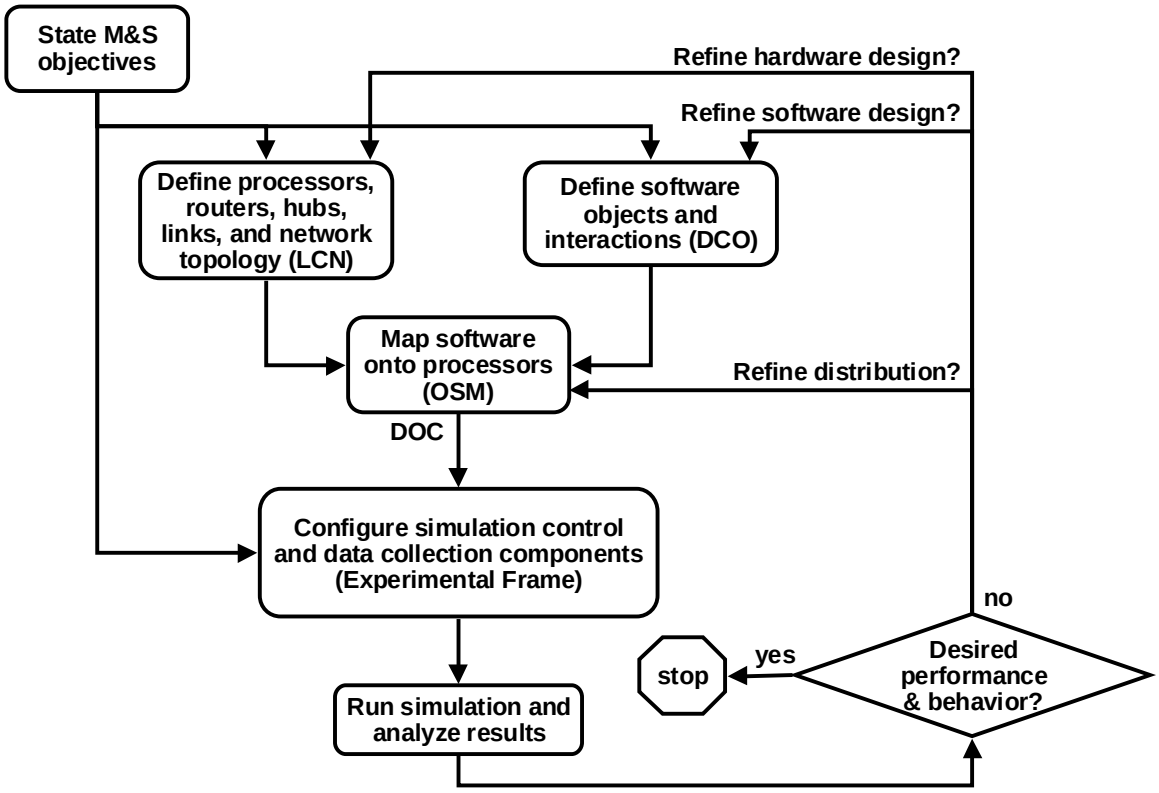


FIGURE 9. DEVS-DOC Modeling and Simulation Process

This process and the DEVS-DOC environment enable modeling the behavior of the software components independent of modeling the computing and networking hardware components. The resulting software and hardware components are then coupled together to form a dynamic model of the distributed object computing system of

interest. Software applications are modeled in the DCO following a distributed object paradigm. Hardware for networked computing components is modeled in the LCN. Mapping DCO software objects onto LCN processors creates the desired DOC model. Adding experimental frame components (acceptors, generators, transducers) prepares the model for simulation and data collection. Analyzing the simulation results may drive decisions to try different software distributions, LCN structures, or DCO configurations. This process is depicted as a flowchart in Figure 9.

Worth noting is the degree of modularity and independence between the LCN, DCO, and OSM representations, as implied within Figure 9 by the iteration loops that refine a DOC model. The DEVS-DOC modeler may choose to evaluate alternative software distributions across processors within the OSM representation reusing the same LCN and DCO models. Alternatively, the modeler may choose to simulate and analyze different LCN topologies and, thus, have no need to rework or modify the DCO or OSM models. Similarly, the modeler may refine and simulate various DCO configuration and interaction structures with no need to revisit the LCN or OSM. However, if a refinement results in the addition or omission of an LCN processor or a DCO software object, obviously, the OSM will also require attention to add or adjust pertinent software - processor mappings.

4.2. Loosely Coupled Networks

The LCN hardware architecture is described in terms of processors, gates, and links. LCN processors are capable of performing computational work for DCO software

objects and transmitting and receiving software object interaction messages. LCN gates interconnect two or more network links and operate in one of two modes: hub or router. As a hub, packets from one link are broadcast out on every other link. As a router, packet address information is used to make routing decisions and switch a packet down a specific link. LCN links provide a communication path between processors and gates.

4.2.1. LCN Links

LCN links describe the communication medium that interconnects two or more LCN nodes. Butler's specification for a link is an object with an assigned error coefficient and a set of one or more channels, where the bandwidth on each link is a random variable to account for servicing actions external to the scope of the simulation space. Our implementation extends on this concept with the specification of technology specific links. In [Hil98a and Hil99] we developed a model of an ethernet link. This ethernet link model allows us to simulate and evaluate DOC systems that employ ethernet technologies within the networking topology.

4.2.2. LCN Gates — Hubs and Routers

LCN gates are objects that interconnect two or more network links. Behavioral attributes for these gates are mode (hub or router), bandwidth for processing traffic, and a buffer size for queuing traffic. The key distinction between router and hub mode is routing decision logic. In hub mode, the gate broadcasts traffic out on each incident link; while in router mode, the gate routes traffic only on the incident link necessary for end-to-end communication.

Router models require routing decision logic to select an outgoing link for end-to-end communication. We opted to not burden the DOC modeler with developing and defining routing tables. Our desire to support technology specific link models has also required the development of technology specific models of media access units (MAUs) or network interface cards (NICs). To separate these two concerns – routing and technology specific NICs – we choose to implement LCN gates as separate object classes. For hubs, a technology specific NIC model class is implemented for each corresponding link technology model. For a router, a single router class model is implemented with the required routing logic. To couple a router to a technology specific link, a DEVS coupled model is used to define the LCN gate. To model a router interconnecting two ethernet links and a T1 carrier link, as in the Figure 3 gate G_0 example, the DEVS coupled model of the gate is composed from two ethernet hub models and a T1 hub model as depicted in Figure 10.

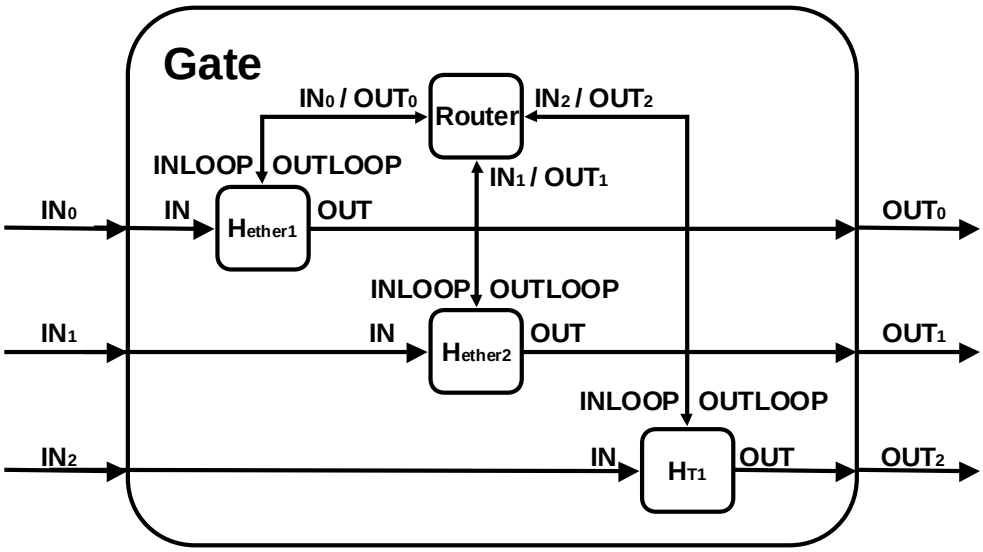


FIGURE 10. DEVS Coupled Model of Figure 3 G_0 Gate

Populating routing tables within LCN routers requires knowledge of the LCN topology and the mapping of DCO software objects to LCN processors. To avoid burdening the DOC modeler with developing this information and maintaining consistency with alternative LCN topologies and alternative DCO to LCN mappings, *route discovery logic* is encoded into the DEVS atomic router model and into the DCO software object model. When a DEVS-DOC simulation starts, the DCO software objects announce themselves, via DEVS messages, to their assigned processors. The processors then broadcast this information on their assigned LCN links. LCN routers receive these broadcasts, load their routing tables, and forward the broadcast. In this way, all LCN routing components learn which incident LCN links service each DCO software object.

4.2.3. LCN Processors

The LCN processor object is capable of performing computational work for DCO software objects and it enables these software objects to interact via the LCN. Butler identifies only two specification parameters for a processor – a computational speed and a storage capacity. If a processor is connected to multiple LCN links, routing decision logic is necessary and we have elected to implement it within the processor. Butler also identified a need to define communication modes – segmenting interaction arcs into packets, dealing with packet overhead, packet acknowledgments, and time-outs – in the OSM abstraction. Again, we have opted to implement these mechanisms within the LCN processor model.

The resulting LCN processor is implemented as a DEVS coupled model of a central processing unit (CPU), a router, and a transport component. The router component is simply a reuse of the router class model just described as part of an LCN gate. Technology specific NIC models are coupled to the processor as needed for the LCN link technologies and topology. The couplings for this DEVS coupled processor model are depicted in Figure 11.

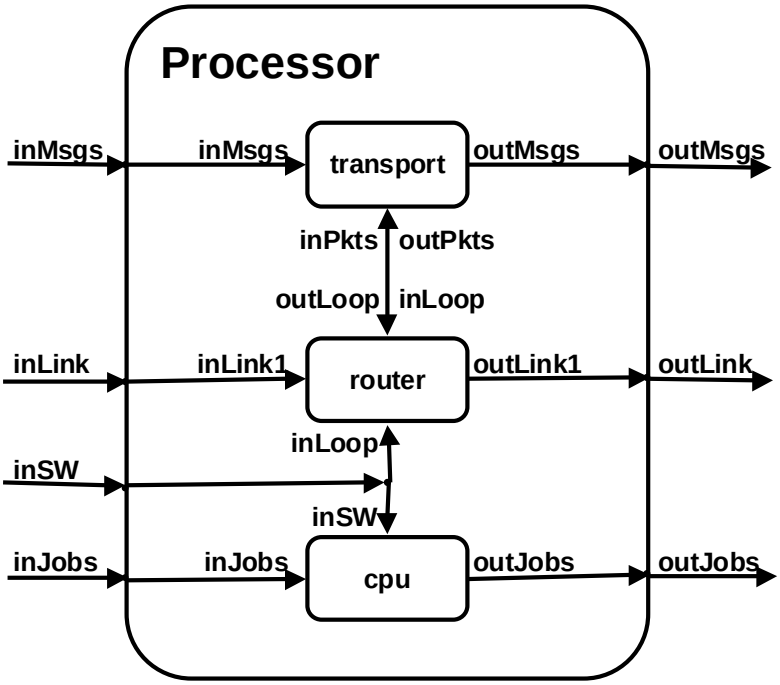


FIGURE 11. DEVS Coupled Processor Model

The CPU is modeled as a DEVS atomic model with two input ports, one output port, six state variables, and three parameters. The *inJobs* input port accepts requests from DCO software objects to execute jobs. The *inSW* input port accepts requests from DCO software objects to load and unload software into, and out of, memory and disk

space. The *outJobs* output port emits jobs that have completed execution. A *cpuSpeed* (cycles per second) parameter determines how quickly data processing operations associated with accepted jobs are executed. When processing multiple jobs, the effective *cpuSpeed* is divided equally across each job. Thus, jobs with equal loading factors are completed on a first-in, first-out basis; whereas, in the case of jobs with unequal loading factors arriving at the same time, the smaller jobs will complete first. A *memSize* (bits) parameter defines the memory usage constraint, and triggers job swapping dynamics as the CPU processes jobs. A *swapTimePenalty* parameter specifies a job processing time loading factor when memory usage becomes constrained during simulation runs.

The transport component is modeled as a DEVS atomic model and segments a DCO software object interaction *msg* into packets for transport across the LCN. The transport component at a destination node receives and collects packets. When all packets for an interaction *msg* are received, the destination transport component delivers the interaction *msg* to the destination DCO software object. In future versions of this transport implementation, we plan to support packet acknowledgment schemes and time-outs. With the transport model, we define the communications modes within the LCN, while Butler opted to define them in the OSM. By extending the functional behavior of the transport component, we can define additional communication modes. Added communication modes will be named. Thus, the user/modeler will be able to map DCO invocation and message arcs onto appropriate communications modes by naming the required communication mode during DCO arc declaration. For more detail see the "Invocation and Message Arcs" description in section 4.3.3.

4.2.4. LCN Topology Mappings

Butler defined two distinct function mappings as part of the LCN representation: *processors to links* and *gates to links*. For our DEVS-DOC implementation, these function mappings are implemented directly within a DEVS *coupled* model as internal couplings between the DEVS models representing the processors, gates, and links. For example, the topology mappings of the example LCN structure in Figure 3, may be coded within a DEVS coupled model as:

```
// map processors to links
Add_coupling ( p0,"outLink", n1,"in");  Add_coupling ( n1,"out", p0,"inLink");
Add_coupling ( p1,"outLink", n1,"in");  Add_coupling ( n1,"out", p1,"inLink");
Add_coupling ( p2,"outLink", n1,"in");  Add_coupling ( n1,"out", p2,"inLink");
Add_coupling ( p3,"outLink", n2,"in");  Add_coupling ( n2,"out", p3,"inLink");
Add_coupling ( p4,"outLink", n2,"in");  Add_coupling ( n2,"out", p4,"inLink");
Add_coupling ( p5,"outLink", n4,"in");  Add_coupling ( n4,"out", p5,"inLink");
Add_coupling ( p6,"outLink", n5,"in");  Add_coupling ( n5,"out", p6,"inLink");
Add_coupling ( p7,"outLink", n6,"in");  Add_coupling ( n6,"out", p7,"inLink");
Add_coupling ( p8,"outLink", n7,"in");  Add_coupling ( n7,"out", p8,"inLink");
// map gate0 to links
Add_coupling ( g0,"out0", n0,"in");     Add_coupling ( n0,"out",  g0,"in0");
Add_coupling ( g0,"out1", n1,"in");     Add_coupling ( n1,"out",  g0,"in1");
Add_coupling ( g0,"out2", n2,"in");     Add_coupling ( n2,"out",  g0,"in2");
// map gate1 to links
Add_coupling ( g1,"out0", n0,"in");     Add_coupling ( n0,"out",  g1,"in0");
Add_coupling ( g1,"out1", n3,"in");     Add_coupling ( n3,"out",  g1,"in1");
// map gate2 to links
Add_coupling ( g2,"out0", n3,"in");     Add_coupling ( n3,"out",  g2,"in0");
Add_coupling ( g2,"out1", n4,"in");     Add_coupling ( n4,"out",  g2,"in1");
Add_coupling ( g2,"out2", n5,"in");     Add_coupling ( n5,"out",  g2,"in2");
Add_coupling ( g2,"out3", n6,"in");     Add_coupling ( n6,"out",  g2,"in3");
Add_coupling ( g2,"out4", n7,"in");     Add_coupling ( n7,"out",  g2,"in4");
```

To further simplify the process of defining such couplings for a DOC modeler, the DEVS-DOC *digraphDOC* class extends the DEVS *digraph* class model to provide specific procedures for coupling LCN components. The following code example results in the same topology mappings / couplings as above but with half as many *Add_coupling* statements. Such *aggregated coupling* statements simplify the modeling effort.

```
// map processors to links
Add_coupling_LCNtoLCN ( p0, n1 );
Add_coupling_LCNtoLCN ( p1, n1 );
Add_coupling_LCNtoLCN ( p2, n1 );
Add_coupling_LCNtoLCN ( p3, n2 );
Add_coupling_LCNtoLCN ( p4, n2 );
Add_coupling_LCNtoLCN ( p5, n4 );
Add_coupling_LCNtoLCN ( p6, n5 );
Add_coupling_LCNtoLCN ( p7, n6 );
Add_coupling_LCNtoLCN ( p8, n7 );
```

```
// map gate0 to links
Add_coupling_LCNtoLCN ( g0,0, n0 );
Add_coupling_LCNtoLCN ( g0,1, n1 );
Add_coupling_LCNtoLCN ( g0,2, n2 );
// map gate1 to links
Add_coupling_LCNtoLCN ( g1,0, n0 );
Add_coupling_LCNtoLCN ( g1,1, n3 );
// map gate2 to links
Add_coupling_LCNtoLCN ( g2,0, n3 );
Add_coupling_LCNtoLCN ( g2,1, n4 );
Add_coupling_LCNtoLCN ( g2,2, n5 );
Add_coupling_LCNtoLCN ( g2,3, n6 );
Add_coupling_LCNtoLCN ( g2,4, n7 );
```

4.3. Distributed Cooperative Objects

The DCO software architecture is described in terms of computational domains, software objects, invocation arcs, and message arcs. A computational domain is a set of software objects that comprise an independent executable program. Following the traditional object orientation concept, software objects represent software components composed of data members (attributes) and functions (methods) that operate on the attributes. Invocation arcs define client–server type software object interactions. Message arcs define peer-to-peer type software object interactions.

4.3.1. Computational Domains

Butler had two purposes for defining computational domains: 1) grouping software processes to extract simulation results, and 2) program scheduling. For program scheduling, Butler identifies *initializer* objects (objects that represent the main program) by assigning them a duty-cycle. The duty-cycle triggers multiple executions of the program during a simulation.

In considering these two purposes for domains, the first purpose is really an experimental frame issue while the second issue is a DCO behavior issue. To maintain a separation of these concerns in our DEVS-DOC implementation, we have opted to identify computational domains within the confines of the experimental frame and drop the *initializer* and duty-cycle attributes from its definition. This makes the role of a computational domain in our implementation strictly an experimental frame issue. To address the program scheduling issue, we add a duty-cycle parameter to the definition of software objects. Thus, *initializer* objects within a computational domain are simply those DCO software objects that have a duty-cycle (set to something less than infinity). Consistent with this role shift, we describe our computational domain implementation further in the experimental frame discussion of section 4.5.

4.3.2. Software Objects

DCO software objects represent the interacting processes of executable programs. Within Butler's structure, software objects have four parameters: a thread mode, a memory storage size, a set of method computational workloads, and a set of method

invocation probabilities. The thread mode determines the multi-threading granularity of the object: *none*, *object*, or *method*. The memory storage size represents the total (data and methods) storage requirement of the object when loaded into processor memory for execution of its methods. Each method is represented by a computational workload factor (e.g., processor cycles) and an invocation probability. The invocation probability is an artifact of the quantum modeling approach and represents the probability that an invocation method will call (invoke) that method.

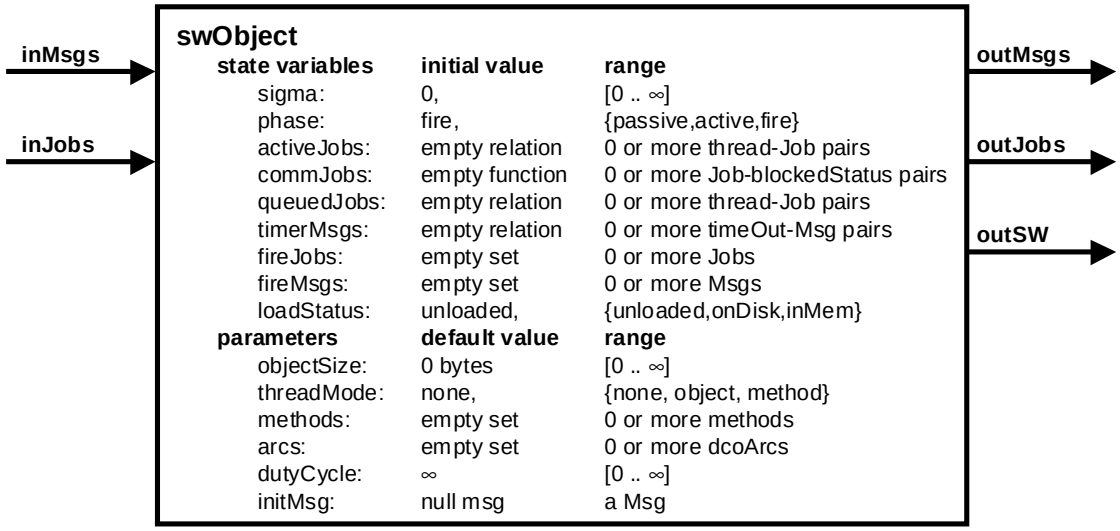


FIGURE 12. DEVS Atomic Model For *swObject*

Our software object (*swObject*) is implemented as a DEVS atomic model. Figure 12 depicts this model, highlighting the inputs, outputs, state variables, and parameters. For state variables, Figure 12 identifies the initial value settings and the range of acceptable values. For the parameters, Figure 12 identifies the default values and range of acceptable values. This *swObject* model extends Butler's software structure of object

size, thread mode, and a set of methods with the definition of a set of interaction arcs, a duty cycle, and an initialization message.

As previously described in “Computational Domains”, our software object representation includes a duty-cycle parameter, which enables identifying and configuring *initializer* objects. The duty-cycle parameter sets the duration between program executions. Setting the duty-cycle to infinity indicates that the object is not an *initializer*. If the DOC modeler is detailing specific method and arc sequences in modeling the DCO, an initialization message (*initMsg*) is also defined to start each program execution with the invocation of a target method.

Counter to Butler's approach of modeling interaction — invocation and message — arcs as DCO components on a level peer to software objects, we define the interaction arcs as components of software objects. In particular, we model interaction arcs as a set of entities that the source software object uses to create messages that are "fired" across the associated LCN during simulations. With this approach, the set of interaction arcs is a parameter in the *swObject* declaration. The implementation of interaction arcs is discussed in more depth in section 4.3.3.

The structure used in defining a set of methods for a software object depends on the DOC modeler’s intent and desired level of detail in modeling the selection and sequencing of methods and interaction arcs. Following the quantum (probabilistic) approach, methods are defined as a set of computational loads paired with a set of invocation probabilities. This implementation satisfies the argument that, in a quantum sense, it is irrelevant which method is invoked by an interaction arc, only that each

method of the object is invoked in correct proportion to the aggregate invocations of all methods of the object. We have also extended this implementation to enable direct modeling of specific sequences of methods and interaction arcs. In particular, interaction arcs can be defined to call (invoke) a specific method on a targeted software object, and the targeted software object method can also declare a specific computational workload and interaction arc firing sequence. These extensions offer a DOC modeler the ability to do directed simulations of specific real-world software interactions. This *quantum* and *directed* sequencing approach to modeling method behavior results in two constructor types.

The constructor for a method defined under the quantum approach has three parameters: a *name*, a *workload*, and an *invocation probability*. The method *name* simply provides a means to identify the method. The *workload* parameter is used to define LCN processor computational loads for jobs that get processed when the method is selected for execution. The *invocation probability* identifies the relative probability that this method is invoked whenever the software object receives an invocation or message. For example, in the Email Application case study discussed in Chapter 6, the email client software objects have two methods: sendMail and resolveNames. Both methods are defined with a workload of 400000 computational cycles and with invocation probabilities of 30% and 70% respectively. The email client set of *methods* is specified with the following three Java statements:

```
methods = new set();
methods.add(new method("sendMail",    400000, 30));
methods.add(new method("resolveNames",400000, 70));
```

At first glance, the constructor for a method defined under the directed sequencing approach appears simpler, but it is actually a bit more complex. The simplistic view is that the method is constructed with a method *name* and a *task* queue. Again, the *name* provides a means to identify the method. The *task* queue defines a sequence of computational workloads and interaction arcs. The software object uses the computational workload to construct a job for the LCN processor to execute, and then fires the associated interaction arc once the LCN completes the job. Then the next workload in the *task* queue is used to construct the next job in the sequence. For example, in the Distributed Federation Simulation case study in Chapter 6, the federate executive (fedex) software object has a single method called "run()" with a task queue (fedexCycle) to model the repetitive cycling of *time-advance-grant* messages from the fedex to the other distributed federate software components. This fedex method set is defined with two statements.

```
methods = new set();
methods.add(new method("run()",   fedexCycle  ));
```

The definition of the fedexCycle task queue is a bit more complex. The fedexCycle has four key steps to compute the *next-time-advance-grant* and three intermediate *time-advance-grants* for an interleaved DEVS cycle [Zei99c]. The task queue is configured to run this cycle for a specified number of iterations, which results in the following Java code to define the fedexCycle task queue.

```
queue fedexCycle = new queue();
for (int i=0; i<Iterations; i++) {
       fedexCycle.add(new task(timeAdvGrant_workload, arc_timeAdvGrant_n));
       fedexCycle.add(new task(timeAdvGrant_workload, arc_timeAdvGrant_n_1));
       fedexCycle.add(new task(timeAdvGrant_workload, arc_timeAdvGrant_n_2));
       fedexCycle.add(new task(timeAdvGrant_workload, arc_timeAdvGrant_n_3));
}
```

4.3.3. Invocation and Message Arcs

Butler defines invocation and message arcs independent of computational domains and software objects. In defining the DCO representation, mapping functions are used to relate a source (calling) software object to each arc and to relate one or more target (called) software objects to each arc. Both arc types are defined by a firing frequency parameter and a message size parameter. The firing frequency is expressed as the amount of computational progress to be made by the source software object between arc firings (software object interactions). The message size parameter represents the number of bytes sent across the LCN. Invocation arcs are further defined with two additional parameters, a return message size and a blocking mode. As invocation arcs represent a client–server interaction, the return message size represents the number of bytes returned by the target software object at the completion of its method execution. The blocking mode parameter is set as either *synchronous* or *asynchronous*. A *synchronous* setting causes the source software object to be blocked from continuing method computation, while an *asynchronous* setting allows method execution to continue at the source software object.

To reduce the number of DCO modeling constructs, we implement a single *dcoArc* class as an extension of the DEVS *entity* class to represent both types of interaction arcs. Collections of such arcs are defined within a DEVS *set*, which serves as a parameter to the source software object and defines the arc to target object mapping. Using the *quantum* modeling approach, an arc is declared by specifying an arc name, a target set of software objects, a message size, a return size, a message type (synchronous, asynchronous, or message), and a firing frequency. As an example, in the Email Application case study discussed in Chapter 6, email clients make invocation queries to the name server object. This name server invocation arc is defined as:

```
//arc to query NameServer
//arc: name, dstList, msgSize, returnSize, msgType, firingFreq
dcoArc query=new dcoArc("queryName","NameServer",100,200,"invokeSync",200000);
```

Using a *directed* modeling approach, the interaction arc constructor drops the "firing frequency" parameter and adds a "called method" parameter. The "firing frequency" is no longer needed as arc firing is made an explicit part of the task queue defined within a *method*, and the "called method" parameter explicitly identifies the *method* to be executed on the targeted software object.

4.3.4. DCO Mappings

Bulter defined five distinct function mappings as part of the DCO representation. As our implementation has aligned the need for computational domains with experimental frames, the mapping of DCO software objects into computational domains

is also shifted into the experimental frame representation. The remaining four function mappings relate arcs to source and destination software objects. As detailed in the preceding two sections, the mapping of arcs to source objects is part of the *swObject* class declaration, and the mapping of arcs to destination objects is part of the *dcoArc* class declaration.

4.4. Object System Mapping

The OSM representation assigns DCO software objects to LCN processors. As depicted in Figure 12, each swObject has two input ports and three output ports. Each of these ports is coupled to input and output ports on the assigned LCN processor. These couplings can be defined within a DEVS *digraph* model with five *Add_coupling()* statements for each assigned swObject. To simplify this for a DOC modeler, the DEVS-DOC *digraphDOC* class extends the DEVS *digraph* class to implement an *Add_coupling_swObject_to_processor(swObjects, processor)* statement. This single statement implements all five of the needed couplings and can accept the *swObjects* argument as a reference to a single DCO swObject or a set of DCO swObjects being mapped to the processor. For example, when the DEVS *set* container *swObjects* contains *swObject1*, *swObject2*, and *swObject3*, the statement

```
Add_coupling_swObject_to_processor( swObjects, processor );
```

adds all the needed couplings as depicted in Figure 13. In particular, this statement couples the *outMsgs* port of each component in *swObjects* to the *inMsgs* port of the *processor*. Similar couplings are made for *outJobs* to *inJobs* and for *outSW* to *inSW*. The

statement also couples the *outMsgs* port of the *processor* to the *inMsgs* port for each component in *swObjects*. Similar couplings are made, as well, for *outJobs* to *doneJobs*. This *aggregated coupling* statement significantly simplifies the modeling effort.

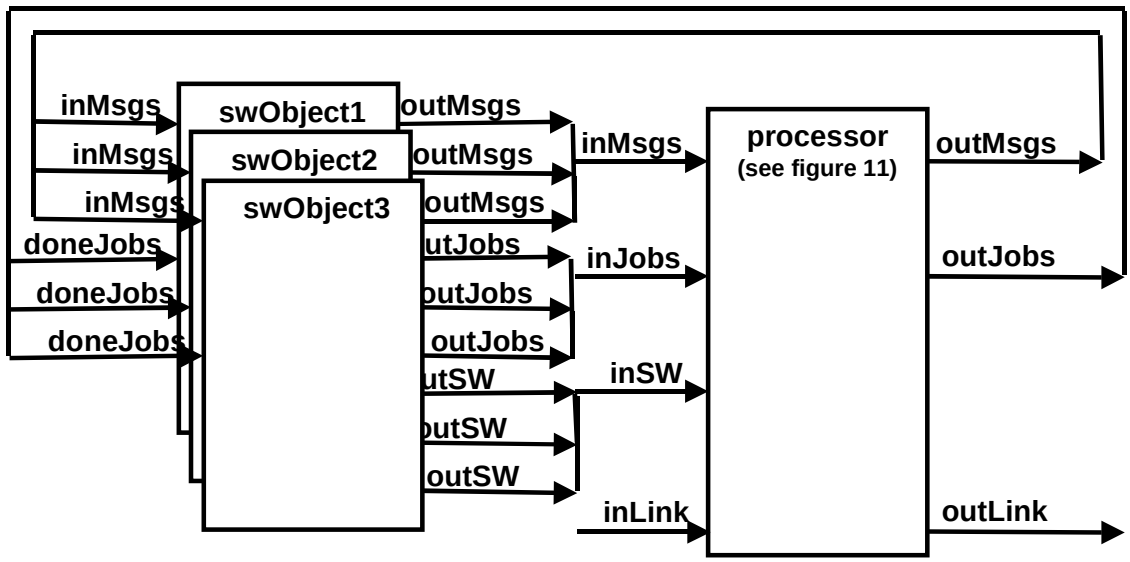


FIGURE 13. DCO Software Objects and LCN Processor Couplings

These couplings and mappings facilitate the following dynamics and interactions. A software object is invoked by receiving a *msg* on its *inMsgs* port, where the *msg* contains the name of the software object in its destination address list. Once invoked, the software object loads itself into processor memory by sending a load software message to the processor *inSW* port. The invocation also causes a method of the software object to be selected for execution, and a computational *job* is sent to the processor *inJobs* port. Once the processor completes executing the *job*, the *job* is returned to the software object via the *outJobs* port of the processor. Each software object receives the *job* completed by the processor and must check whether the *job* on the *doneJobs* port originated from itself.

If so, the software object marks computational progress, selects a *dcoArc*, creates a *msg*, and sends it to the processor port *inMsgs* for transmission across the LCN. When the processor receives a *msg* destined for one of its associated software objects, that *msg* is sent on the *outMsgs* port. Again, each mapped software object receives the *msg* and must verify whether it is the targeted (destination) software object prior to servicing the *msg*.

At this level of abstraction, the DCO to LCN mapping may be more appropriately characterized as a software to firmware mapping in that the processor components represent an aggregation of hardware and software functional resources. The LCN *cpu* actually represents a combination of processor subsystems: central processing unit, disk and system memory system, system bus and input/output ports. Likewise, the gate actually represents a combination of network interfaces and intra-processor communication channels for co-hosted interacting software objects.

4.5. DEVS-DOC Experimental Frame

For distributed object computing systems, Table 1 depicts a set of metrics to observe and assess dynamics and behaviors in running a simulation of a DOC model. Table 1 is revised from [But94] and lists some of the major sets of information metrics that may be derived from simulation of the DCO and LCN interactions. Metrics marked with a "∗" are applicable for components of that class. Metrics marked with a "Σ" indicates that an aggregation of the metric is applicable over the set of components in that class. For example, the "Σ" under the *Domain* DCO Class for the metric *Computational Work Performed* indicates this statistic is a summation of the computational work

completed for each software object within the domain. These information elements define the objectives for the distributed computing system modeling and simulation enterprise and specify the major functional components for the experimental frame.

	LCN Classes			DCO Classes		
Description of Metric	***Processor***	***Gate***	***Link***	***Domain***	***Object***	***Interaction Arc***
Computational Work Performed	*			∑	*	
Active Time/Utilization	*			*	*	
I/O Data Load	*			∑	*	
Utilization of Storage	*					
Percentage of Active Objects	*			*		
Degree of Multithreading	*				*	
Length of Execution Queues					*	
# of Initialization Invocations				*	*	
Total Execution Time				*	*	
Coefficient of Interaction				*		
Data Traffic		*	∑			*
Utilization of Bandwidth		*	∑			
% of Packet Retransmission			∑			*
Length of Packet Buffer		*				
Net Throughput of Data		*	∑			*
Rate of Overhead		*	∑			*
Gross & Net Response Time						*

TABLE 1. Major DOC System Simulation Metrics

This research effort has developed a set of transducers for the processor, link, domain, object, and the interaction arc classes listed in Table 1. The current link transducer is specifically for ethernet links. Development of a gate (hub and router) transducer and a generic link transducer are planned as part of future directions.

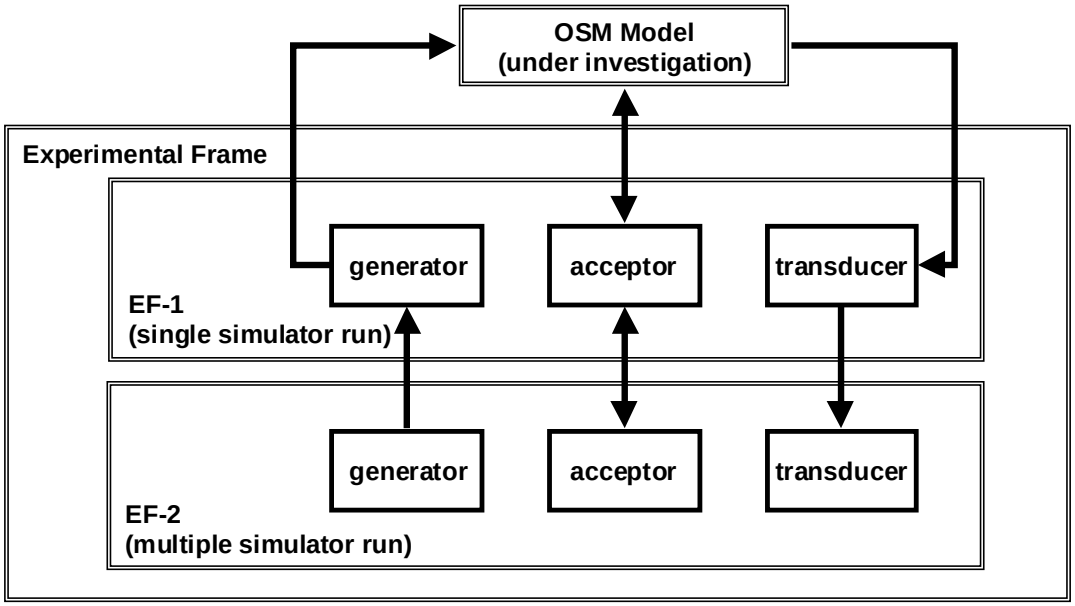


FIGURE 14. Layered Experimental Frame

Given the number of random variables in a DOC model, we have also developed a *tuples* transducer to collect results across multiple simulation runs. The *tuples* transducer collects the results of each transducer from a single simulation run and computes the mean, variance, lower and upper bounds for the metrics of Table 1. This approach can be viewed as an experimental frame containing two operational layers: the first layer to support individual simulation runs, and a second layer to drive multiple simulation runs and aggregate the results into statistical quantities. A traditional experimental frame (EF-1) is constructed with a generator to stimulate the model, an acceptor to control the simulation, and a transducer to collect simulation data and summarize the results. The second operational layer experimental frame (EF-2) stimulates, controls, collects and summarizes the results of the first experimental frame layer. Figure 14 depicts this layered experimental frame concept.

4.5.1. Computational Domains

As pointed out in the DCO Computational Domains section, the prime purpose for defining computational domains is to group related software processes for extraction of simulation results across the aggregate. For this reason, we implement the definition of computational domains as part of the experimental frame. The implementation is a DEVS *set* class. A *set* is declared for each computational domain being represented, and appropriate swObjects are added to each domain (*set*). The *set* of software objects is then used in declaring the domain transducer that monitors the software objects of the domain during simulation.

4.5.2. DOC Transducers

This research has implemented a transducer for the DOC classes of *domains*, *swObjects*, interaction arcs (*msgs*), *processors*, and *ethernet* links. Implementing a transducer for gates and generic links is planned for future efforts. Each of these transducers is implemented as an extension of the DEVS *atomic* class. As previously described, a *tuples* transducer class is implemented to collect and compute the mean, variance, lower and upper bounds for these class specific transducers. The *tuples* transducer is also a child of the DEVS *atomic* class. To simplify the coupling of these transducers to DCO and LCN components, the *digraphDOC* class provides procedures that aggregate the otherwise multiple *Add_coupling()* statements to a single `Add_coupling_transducerCLASS(monitoredObjects,interactingObjects,transducer);` statement.

4.6. DEVS-DOC Class Hierarchy

In section 3.2.2, DEVS Object-Orientation, we provide a short overview of the DEVS class hierarchy. In implementing the DEVS-DOC modeling and simulation environment in DEVSJAVA, we have extensively used the object-orientation property of inheritance to simplify the implementation effort. To realize LCN, DCO, and experimental frame component classes, we implement them as extensions of the DEVS *atomic* class. The one exception is the LCN *processor* class, which is implemented as an extension of the *coupled* class, as the processor model is a composition of the DEVS *atomic* model for LCN *cpu*, *router*, and *transport* components. This inheritance hierarchy is depicted in Figure 15.

Within the DOC model, as devised as part of this research, several inert or passive components are characterized — e.g., *dcoArc* for invocation and message arcs, and *job* for cpu workloads. To simplify implementing these DOC specific passive components, we extend the DEVS *entity* class to create the needed classes in the same fashion as the HCCL outlined in section 3.2.2. Figure 16 depicts the inheritance hierarchy for these passive DOC container components.

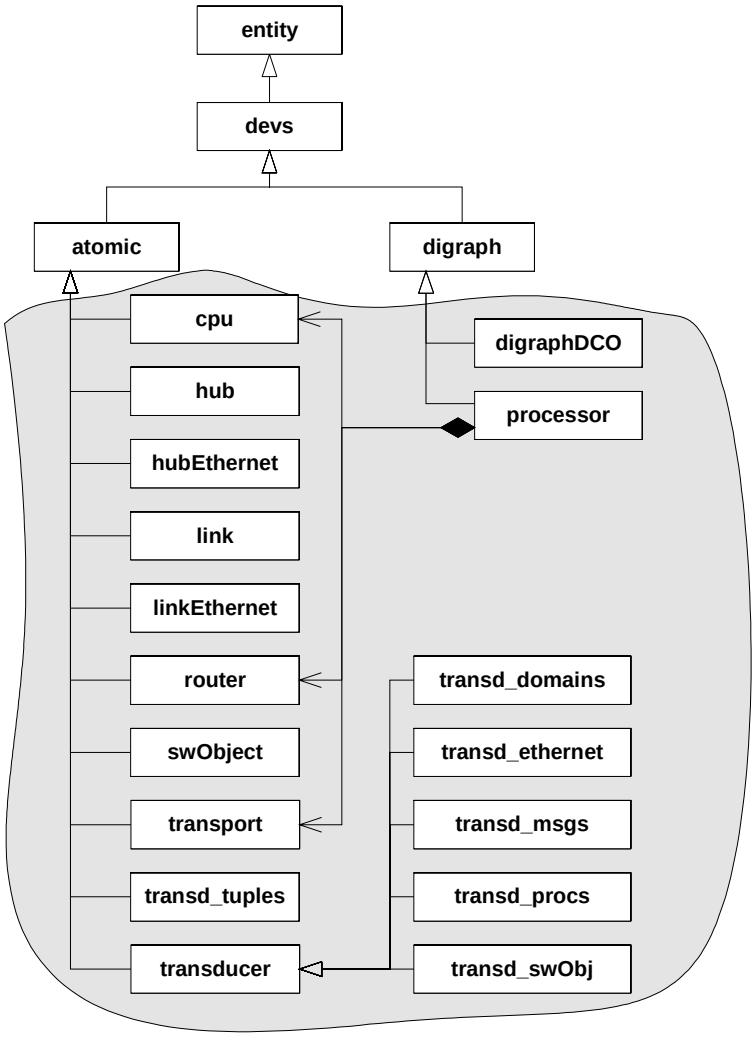


FIGURE 15. DEVS-DOC Class Hierarchy

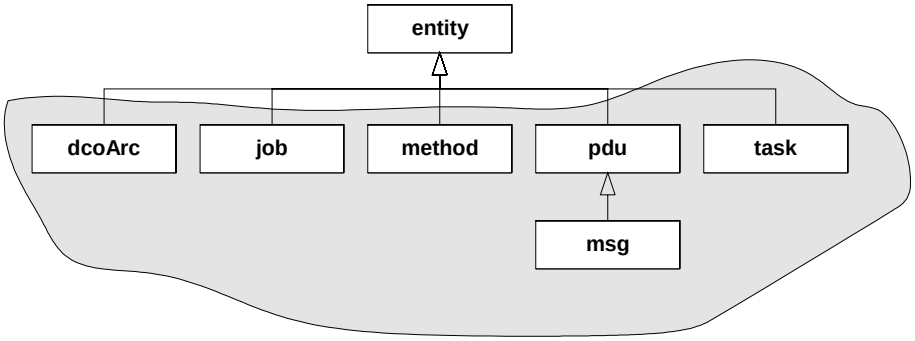


FIGURE 16. DEVS-DOC Container Class Hierarchy

5. DEVS-DOC COMPONENT BEHAVIOR

In section 4, we detailed the structural representation used to model distributed object computing systems. In this section, we provide behavioral specifications for the various DEVS-DOC components. In particular, we focus on defining the input events, states, state transitions, outputs, output functions, and time advance functions needed to implement the dynamics of the various DEVS-DOC components. These behavioral representations were developed and refined throughout the research effort behind this dissertation. The Parallel DEVS formalism is utilized to succinctly and rigorously define these dynamics.

5.1. LCN Component Behavior

The overall behavior of an LCN model is an aggregation of the individual LCN component behaviors, which form the LCN. The behavior of the individual components within the LCN model, however, is constrained by the couplings that define the LCN composition and topology. In this section, the dynamic behaviors developed for the atomic LCN components are defined.

5.1.1. Ethernet Link

The LCN *link_ethernet* model represents the ethernet cable interconnecting two or more devices. The basic behavior required is to receive transmitted messages — *data* frames — from connected nodes; after an appropriate propagation delay time, output a *preamble* to signal the start of a frame; and then after an appropriate transmit delay time,

output the transmitted *data* frame to signal the end of the transmission. If any new frames are received during this process, a collision has occurred. In the case of a collision, after the propagation delay time the model outputs a *null* frame, which signals the collision event to all connected nodes. On detection of a collision, the ethernet standard requires nodes to emit a *noiseburst*. So after a collision and the appropriate propagation delay time, the ethernet link model outputs a *noiseburst*. For connected nodes, the *preamble* output signals that the ethernet is busy, the *null* frame output signals a collision, and a *data* or *noiseburst* frame output signals the end of a transmission, i.e., the ethernet is idle.

The dynamics of this model need to account for the propagation delay of transmissions from one node to all other ethernet connected nodes. The propagation delay is the time for a signal from one node to reach another node, which depends on cable length. To keep it simple, we assume a worst case propagation time between all nodes. Thus, we only need a single input port to receive transmissions, a single output port to broadcast the signal, and a single time delay state variable to track the delay for all connected nodes. From the IEEE 802.3 standard, an ethernet may have one to five 500 meter segments (using four repeaters) with the worst-case propagation delay per segment being 25.6 micro-seconds.

To further simplify the model and enhance simulation performance, our model also accounts for the time to transmit each frame. Thus, an input message from a node is a pair of values representing the *data* frame to be transmitted and the time to transmit the *data* frame — the time between putting the first and last bits of the *data* frame on the

ethernet. This approach improves model simulation performance by only requiring a single message exchange from a transmitting node model to the ethernet link model rather than two exchanges — one for the *preamble* (start of transmission) and one for the *data* frame (end of transmission).

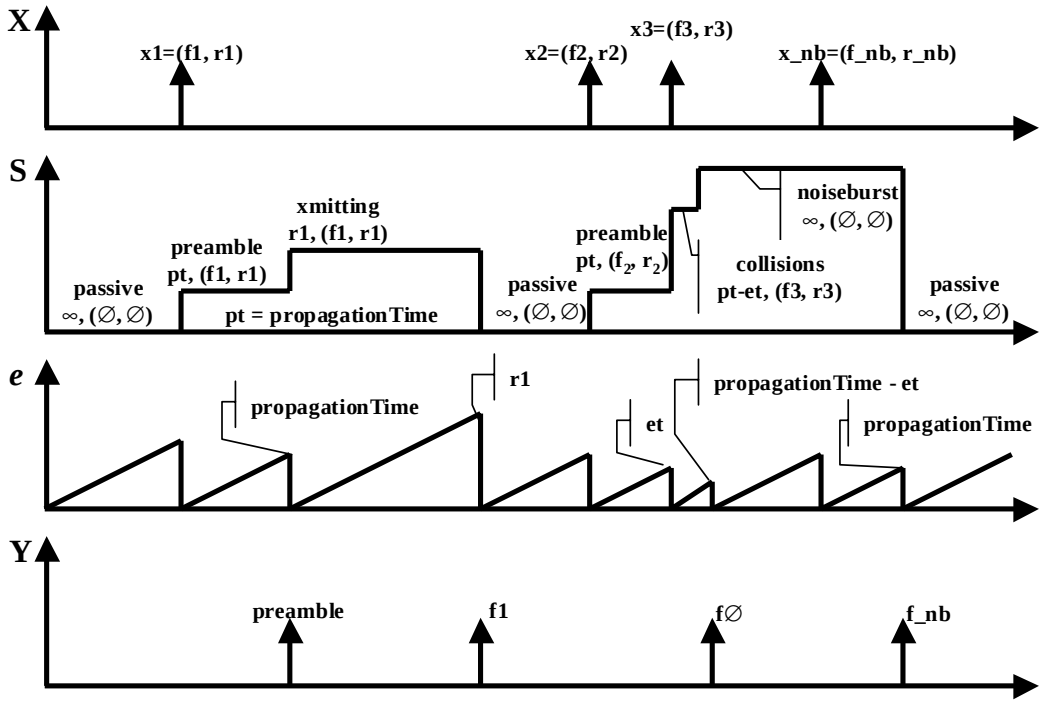


FIGURE 17. LCN Ethernet Link Discrete Event Time Segments

The discrete event time segment trajectories for the required dynamics of this ethernet link model are depicted in Figure 17. The dynamics are plotted over time and represent the input events, **X**; the model state changes, **S**; the elapsed time in a given state, ***e***; and the model output events, **Y**. The depicted dynamics start with the ethernet in a "passive" state with the time advance function set to infinity (∞) and the pair of values representing the data frame being transmitted and the time to transmit both set to null.

The next event is the input event "x1" that contains a pair of values, f1 being a data frame and r1 being its transmit time. The "x1" input event sets the model into a "preamble" state with the time advance function set to the propagation time (pt) and the frame and transmit time pair set to (f1,r1). With the elapse of the propagation time, the model has an internal event that causes the output of a *preamble* and sets its internal state to "xmitting" with a time advance of r1. With the elapse of r1, the internal event causes the output of the data frame f1 and sets the model back to its "passive" state. The next event sequence depicts the progression of a collision scenario.

A Parallel DEVS representation for this LCN *link_ethernet* model follows. This representation is also included in Appendix C, DEVS-DOC Behavioral Specifications.

$DEVS_{link\_ethernet} = < X, Y, S, \delta_{int}, \delta_{ext}, \delta_{conf}, \lambda, ta >$, where

InPorts = {"in"}
OutPorts = {"out"}
$X = \{(f,r) \mid f \in F, r \in \Re\}$
$Y = \{\text{preamble, noiseburst}\} \cup F$
$S = \{\text{"passive","xmitting","collisions","noiseburst"}\} \times \Re_0^+ \times X$

$\delta_{ext}((\text{phase},\sigma,x_{last}),e,(\text{"in"},x)) =$ case phase is

("preamble",propagtionTime,x)	"passive"
("collisions",σ-e,x)	"preamble"
("collisions",σ-e,x)	"xmitting"
("collisions",σ-e,x)	"collisions"
("noiseburst",propagationTime,x)	"noiseburst"

$\delta_{int}(\text{phase},\sigma,x_{last}) =$ case phase is

("passive",∞,($\varnothing$,$\varnothing$))	"passive"
("xmitting",r_{last},x_{last})	"preamble"
("passive",∞,($\varnothing$,$\varnothing$))	"xmitting"
("noiseburst",∞,($\varnothing$,$\varnothing$))	"collisions"
("passive",∞,($\varnothing$,$\varnothing$))	"noiseburst"

$\delta_{conf}(s,ta(s),x) = \delta_{ext}(\delta_{int}(s),0,x)$

λ (phase,σ,x_{last}) = case phase is

("out",preamble)	"preamble"
("out",f_{last})	"xmitting"
("out",$f_{\varnothing}$)	"collisions"
("out",noiseburst)	"noiseburst"

ta(phase,σ,x_{last}) = σ

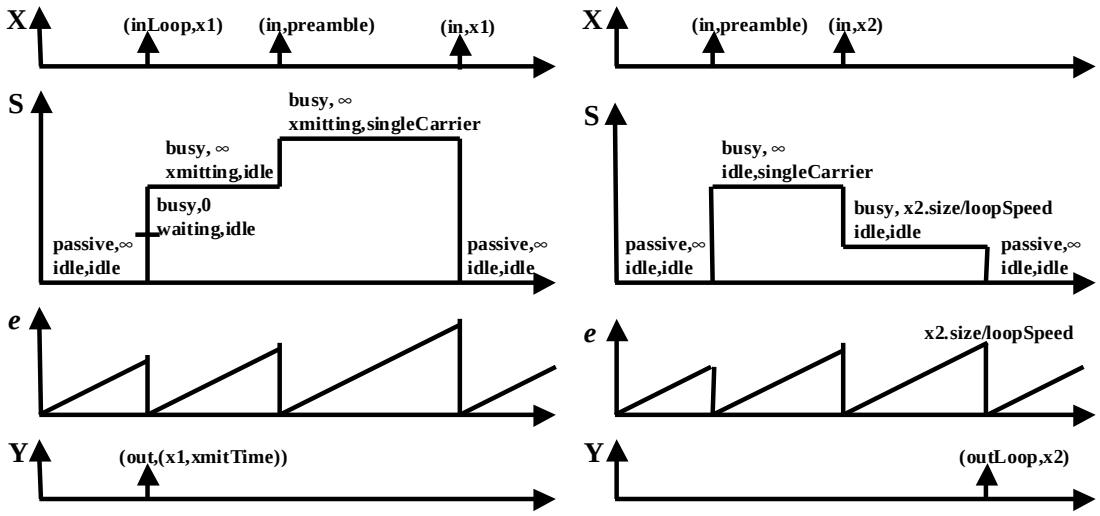
where $x_{last}=(f_{last},r_{last})$ represents the last input pair received
F = frames, to include the null frame $f_{\varnothing}$

5.1.2. Ethernet Hub

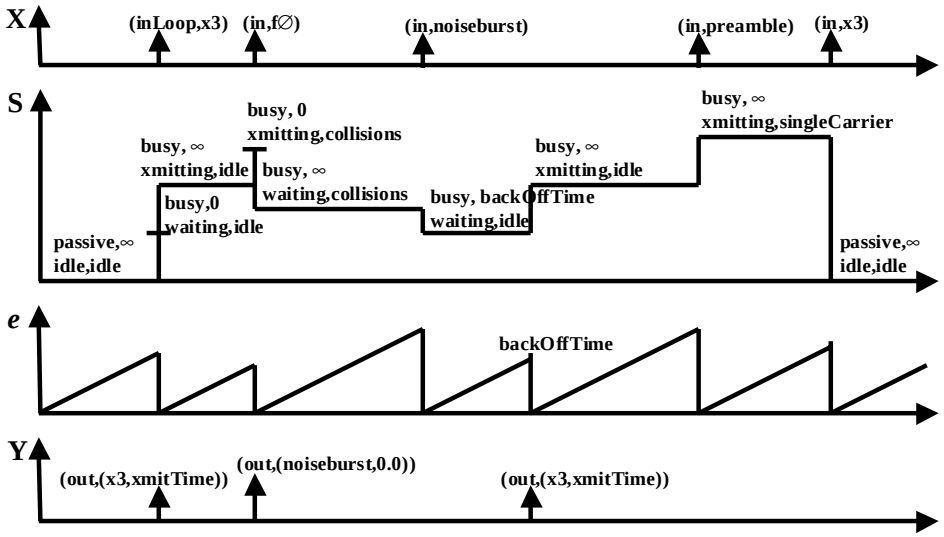
The LCN *hub_ethernet* model represents a multi-port communications device that receives LCN traffic on one port and broadcasts that traffic out on all other ports, where all but one set of ports follow the IEEE 802.3 ethernet protocol. The exception port set, the "inLoop" port and "outLoop" port, provides a means for interconnecting other LCN devices — processors and routers — to ethernet links. The number of ethernet ports modeled is a configuration parameter.

The discrete event time segment trajectories for three scenarios of a single ethernet hub system are depicted in Figure 18. In the first scenario, traffic is received on the local loop port while the ethernet is idle; the model immediately sends the traffic out over the ethernet; and no collisions are detected. In the second scenario, traffic is received from the ethernet and is immediately sent out the local loop. For the third scenario, local loop traffic is again sent over the ethernet; however, this time a collision is detected; the model waits, and then tries to send the traffic again; this time successfully.

A complete specification of the LCN *hub_ethernet* dynamic behavior is provided in Appendix C. To simplify the specification, a single ethernet port was assumed. Here, we highlight extracts of the Appendix C dynamic behavior specification to embellish on a few concepts.



a. From local loop to ethernet scenario **b. From ethernet to local loop scenario**



c. Collision scenario

FIGURE 18. LCN Ethernet Hub Discrete Event Time Segments

The *hub_ethernet* state space matrix, S, is the cross-product of a number of vectors and scalars. In particular,

$$S = \texttt{Phase} \times \sigma \times \texttt{XmitState} \times \texttt{MediaState} \times \texttt{LoopDelay} \times \texttt{LoopBuffer} \times \texttt{PortDelay} \times \texttt{PortBuffer} \times \texttt{BackOffCount}$$

where the vector *Phase* and the scalar σ represent the two basic state variables of any DEVS model.

The *XmitState* and the *MediaState* are two of the ethernet hub specific vectors. The *XmitState* vector represents the transmission status of the hub on the outgoing ethernet link, i.e., *XmitState* = {idle, waitingForIdle, xmitting}. The *XmitState* is "idle" when the hub has no traffic to send out the ethernet; is "waitingForIdle" when the hub has traffic but, in following the IEEE 802.3 media access protocol rules, is unable to transmit; and, is "xmitting" when the hub is transmitting queued traffic. The *MediaState* vector represents the detected state of the ethernet link, i.e., *MediaState* = {idle, singleCarrier, collisions}. The *MediaState* is "idle" when traffic has not been detected on the ethernet; is "singleCarrier" when the start of traffic (a preamble) has been detected (received); and, is in "collisions" when a collision (a null event) has been detected (received).

The *LoopDelay* scalar represents the time to transmit traffic out the local "outLoop" port; while the *PortDelay* scalar represents the time to wait until attempting to transmit traffic out the ethernet link port "out1". When there is no traffic for these ports, these scalars are set to infinity, ∞.

The *LoopBuffer* and the *PortBuffer* are queues to hold traffic destined for transmission out the local "outLoop" port and the ethernet link port "out1", respectively.

Following the IEEE 802.3 media access protocol, when a device has traffic to send and detects traffic on the ethernet, the device waits (backs off) a random number of time slots before attempting to send its traffic. A time slot is equal to the worst-case propagation time. For each back off, the range of random numbers increases exponentially. E.g., after the first collision, each station waits either 0 or 1 slot times before trying again; the second time, either 0, 1, 2, or 3 time slots; in general, after i collisions, a random number between 0 and 2^i-1. The *BackOffCount* scalar in the state space matrix represents the number of sequential collisions encountered.

The external transition function is defined with four sequential state change steps, which are summarized as follows:

δ_{ext}(s,*e*,(InPorts,X)) =

(,,,,loopDelay-*e*,,portDelay-*e*,,) [5]	before processing input events X
(,,,,,,,P^{+}.add(x,xt),)	for each x event on "inLoop"
(,,xs,ms,ld,lb,pd,pb,boc)	for each x event on "in"
(ph,sigma,xs,,,,pd,,boc)	after processing input events X

As annotated, the first state change — (,,,,loopDelay-*e*,,portDelay-*e*,,) — is executed before processing any of the input events X. This state change step re-computes the *LoopDelay* and *PortDelay* scalars based on the elapsed time, *e*. If either *LoopDelay* or

[5] In this notation, commas are used to signify each of the state variables and any equations within them signify the new value to be assigned. In the (,,,,loopDelay-*e*,,portDelay-*e*,,) case, the first four state variables, along with the sixth, eighth, and ninth state variables, remain unchanged. The fifth state variable is LoopDelay, which is set to a new value of loopDelay-*e*. Similarly, the seventh state variable is PortDelay, which is set to a new value of portDelay-*e*.

PortDelay happen to be at infinity, then the subtraction of any arbitrary elasped time *e* (which is less than infinity) results in infinity. With the completion of this first sequential state change step, these new delay values apply in the remaining three steps.

As annotated, the second sequential state change step (,,,,,,,P^{+}.add(x,xt),) is applied repetitively for each input event from the "inLoop" input port. Likewise, the third step is applied repetitively for each input event from the ethernet "in1" input port. The fourth and final step is applied once after processing all the input events.

Appendix C provides the complete specification with details on the variables used in the sequential state change steps just described, as well as details on the other DEVS functions.

5.1.3. Router

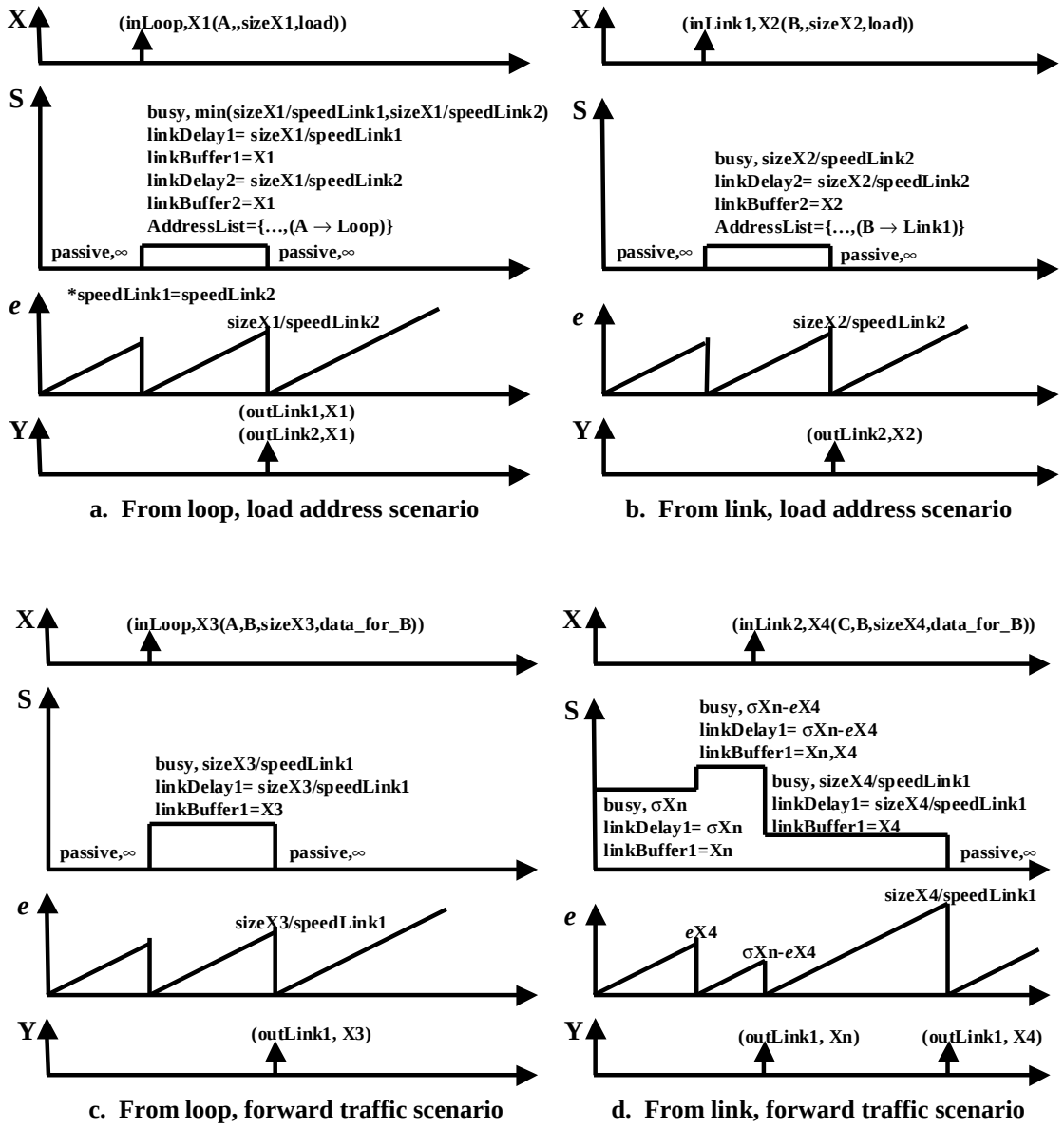
The LCN *router* model represents a multi-port communications device that receives LCN traffic on one port and then, based on its routing table, forwards the traffic over a link heading towards the destination. The *router* has one set of ports, the "inLoop" and "outLoop" ports, to provide a path for interconnecting DCO software objects local to the *router* node to other LCN components via LCN link paths. The number of *router* LCN link ports modeled is a configuration parameter.

To avoid requiring a DEVS-DOC modeler to define routing tables, the *router* model includes dynamic behavior mechanisms that automatically discover the routing information needed to setup a routing table. This route discovery behavior is triggered by a DCO software object behavior that sends a "load" message over the LCN, at the start of

a simulation. On the receipt of this "load" message, the *router* updates its route table by noting the source name and the link servicing it. The *router* then broadcasts this "load" message out all other links to share this knowledge with other LCN components. Within the experimental frame, these route table update dynamics can be defined as a simulation startup sequence that the transducers (data collectors) can ignore.

The first discrete event trajectory plot in Figure 19 depicts the progression of a "load address" event on a "loop" input port. The scenario starts with an input event of message X1 on the "inLoop" port. Message X1 is from DCO software object "A", has a size of "sizeX1", and is a "load" message. The state change reflects X1 being queued for output on outLink1 and outLink2, and the AddressList being updated to associate traffic destined for software object "A" to the "Loop" output port.

In similar fashion, the second event plot in Figure 19 depicts the progression of a "load address" event on a "link" input port. In particular, message X2 arrives on the "inLink1" port and is a "load" message from software object "B". The model queues message X2 for forwarding on the outLink2 port and updates the AddressList to associate traffic for "B" to the "Link1" output port.



a. From loop, load address scenario **b. From link, load address scenario**

c. From loop, forward traffic scenario **d. From link, forward traffic scenario**

FIGURE 19. LCN Router Discrete Event Time Segments

The third and fourth plots in Figure 19 depict traffic reception and forwarding scenarios. In plot c, traffic X3 is from the local loop DCO software object A and is destined to software object B. In this scenario, the *router* is initially passive. In receiving X3, it is queued for "outLink1" (based on the prior occurrence of scenario b),

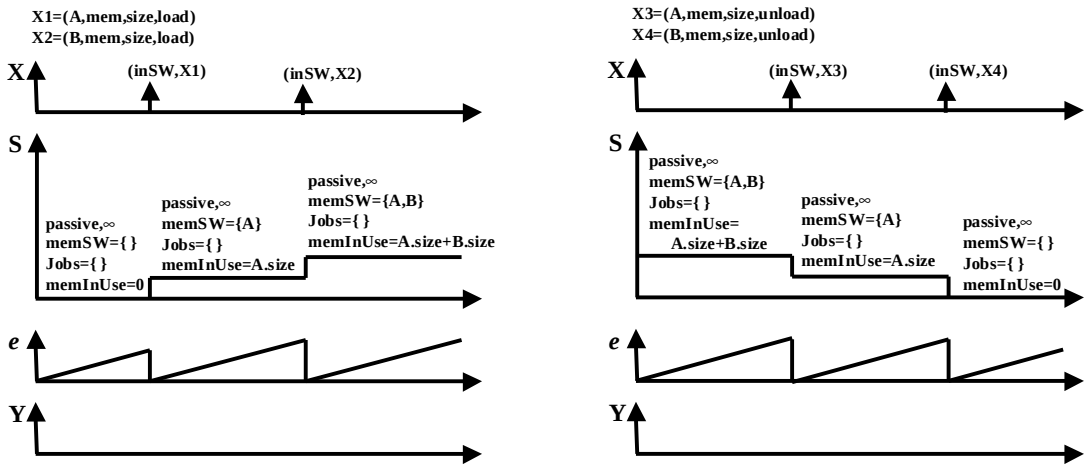
and the next internal event for the *router* is scheduled by setting σ to the transmit time of sending X3 over "outLink1", i.e., the size of X3 divided by the link rate of "outLink1".

Similarly, plot d is the event scenario for traffic received on "inLink2", where X4 is from software object C and destined to B. In this scenario, the *router* is already transmitting traffic over "outLink1" when X4 is received. So, the schedule for completing the current "outLink1" transmission event is reset based on the elasped time *e*X4; and traffic X4 is queued on the "outLink1" buffer. Once, the current "outLink1" transmission completes, the output event of sending X4 over "outLink1" is scheduled based on the transmit time needed for X4.

A complete specification of the LCN *router* behavior is provided in Appendix C. For the external transition function, a sequential set of state change steps is specified in the same style described in section 5.1.2 for the ethernet hub.

5.1.4. Central Processing Unit (CPU)

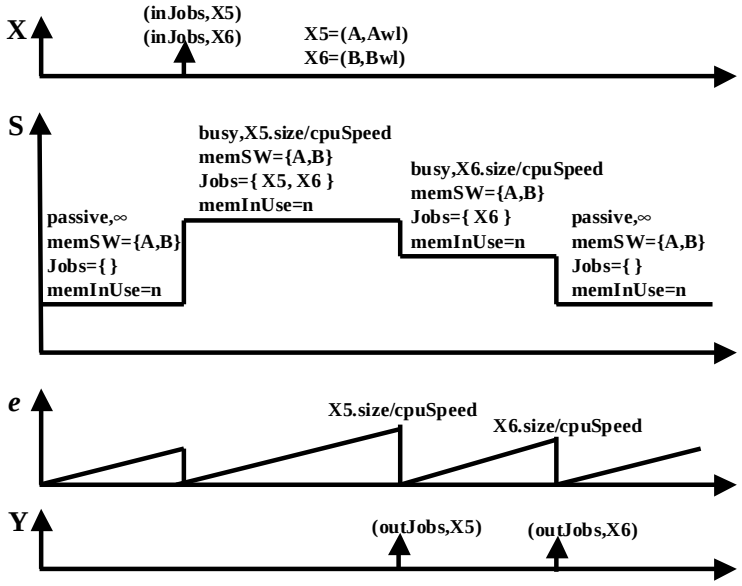
The LCN cpu model represents the behavior of LCN processors as they compute jobs generated from DCO software objects. The model has two input ports; "inJobs" for receiving jobs that are to be processed, and "inSW" for receiving requests to load and unload software in memory. The "memSW" and "memInUse" state variables maintain the status and resource demands of these loads and unloads. Plots a and b in Figure 20 depict event sequence scenarios for such loads and unloads.



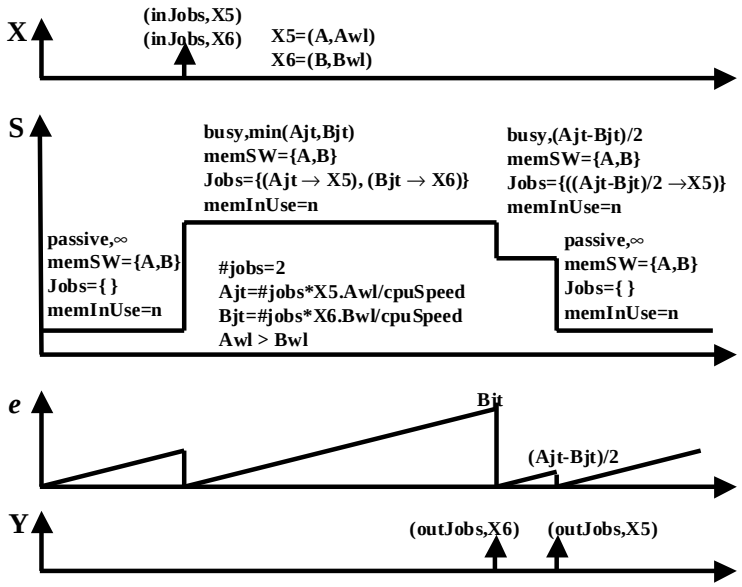
a. LCN cpu load memory scenario **b. LCN cpu unload memory scenario**

FIGURE 20. LCN cpu "inSW" Discrete Event Time Segments

Two cpu model types are available in the DEVS-DOC framework — a single-tasking cpu and a multi-tasking cpu. While the behavior resulting from events on the "inSW" port is the same for both cpu types, the behavior resulting from jobs on the "inJobs" port can vary significantly. The single-tasking cpu accepts and queues multiple job requests, processing them one at a time on a first-in, first-out (FIFO) basis. The multi-tasking cpu accepts multiple job requests and processes all jobs concurrently. The effective cpu speed available to each job, however, is an equally divided fraction of the total cpu speed based on the number of jobs in the cpu.



a. cpu_singleTask job scenario



b. cpu_multiTask job scenario

FIGURE 21. LCN cpu Discrete Event Time Segments

Figure 21 depicts the scenario of two jobs, X5 and X6, arriving at the cpu at the same time. Plot a depicts the scenario for the single-tasking cpu and plot b for the multi-

tasking cpu. In this example, we assume the work load of job X5 is larger than the work load of X6 and that the speed of the two cpu's is the same. The single-tasking cpu treats one of the jobs (X5) as the first arrival, processes it, and outputs it on "outJobs"; the single-tasking cpu then processes job X6 and outputs it. The job outputs of the single-tasking processor are quite staggered in comparison to the multi-tasking cpu. We can also note that while the duration to compute both jobs is equal, the multi-tasking processor completes job X6, with the smaller work load, first.

A specification of behavior for both cpu types is provided in Appendix C.

5.1.5. Transport

The LCN *transport* model is a component to represent behaviors of various communication modes. The current DEVS-DOC *transport* model supports only one communication mode. The required behavior of the model is to partition DCO software object messages into packets for transmission over the LCN, and to reassemble received packets into messages for delivery to software objects. The model receives software object messages of arbitrary length on the "inMsgs" input port; partitions them into packets of a set maximum length; and sends the resulting packets out the "outPkts" port. The model collects incoming packets via the "inPkts" input port; stores these inbound packets in a receiving queue buffer; and when all the packets for a message have been collected, delivers that message to the DCO software object via the "outMsgs" port.

The Parallel DEVS with Ports specification is provided in Appendix C. Figure 22 depicts a discrete event scenario with the *transport* model partitioning a message into packets and with the model receiving a series of packets to form a message.

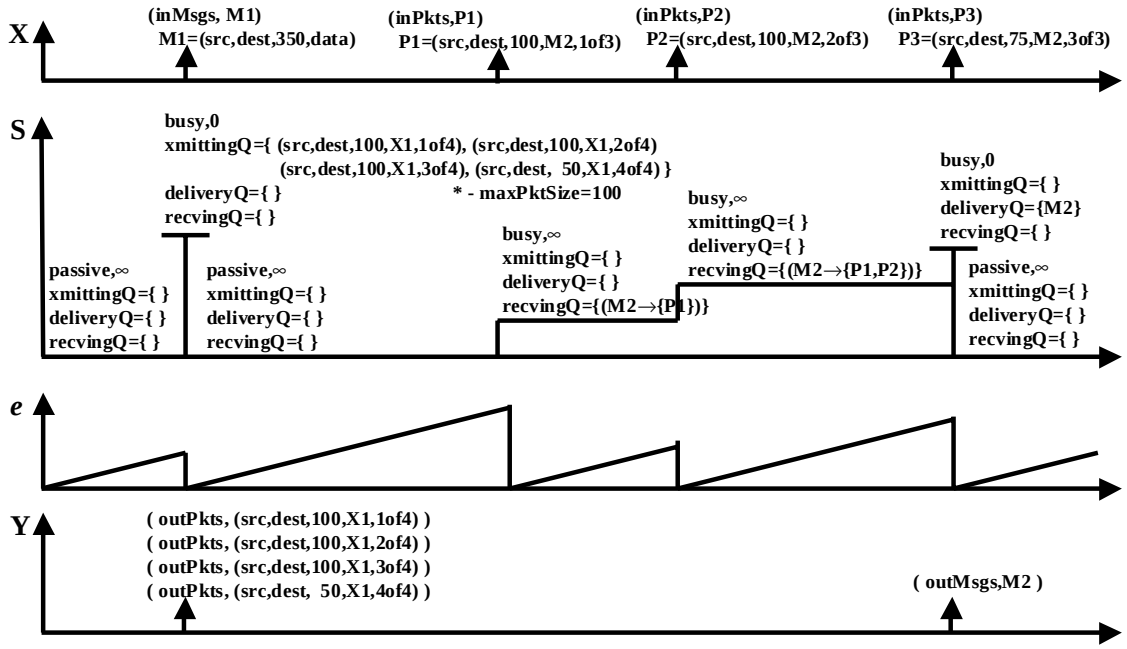


FIGURE 22. LCN transport Discrete Event Time Segments

5.2. DCO Software Object

The only DCO component is the software object model, *swObject*. This model represents software components as the interacting processes that constitute executing programs. The *swObject* model has two input ports, "inMsgs" and "doneJobs". The "inMsgs" port is to receive exchanges (invocations, invocation responses, and messages) from other software objects. The "doneJobs" port is to receive completed jobs from the LCN processor. The *swObject* has three output ports, "outMsgs," "outJobs," and

"outSW." The "outMsgs" port is to send out invocations and messages to other software objects. The "outJobs" port is to send jobs to the LCN processor for execution. And the "outSW" port is to signal the loading and unloading of the software object onto the processor disk and memory resources. Loading the processor disk also triggers the LCN *router* mechanism for automatic route discovery, see section 5.1.3.

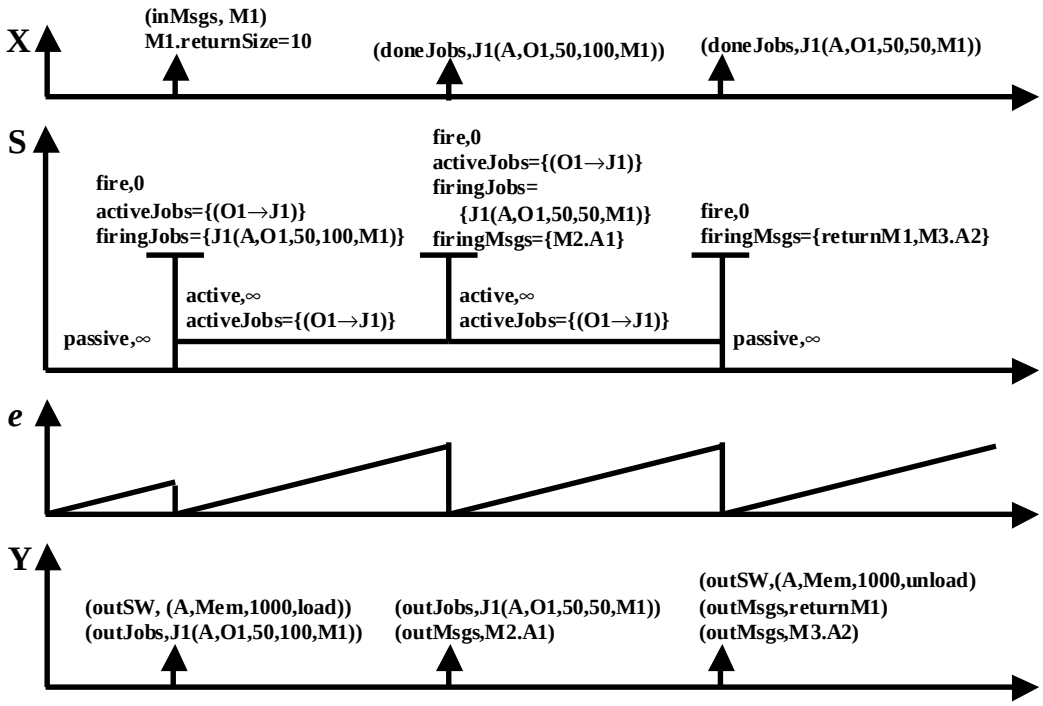


FIGURE 23. DCO swObject Discrete Event Time Segments: Simple Scenario

Figure 23 illustrates a relatively simple event and state transition scenario of a software object receiving an invocation request, selecting the method, processing the job, interacting with other software objects, and then returning a response to the initial invocation. For this simple scenario, the multi-threading mode of the object has no

impacts on its resultant behavior as we have only considered a single invocation. On receiving the invocation message M1, the object selects method O1 (based on the modeler's configuration for the object) to create job J1, and immediately sends two events to the processor: a trigger to load software into memory and the job J1 to execute. The software object is now active and waits ($\sigma = \infty$), for an external event. Note, J1 is configured to execute in two steps, 50 work load units at a time. On receipt of the completion of the first half of J1, the software object selects interaction arc A1 (based on the modeler's configuration for the object) to create an interaction message M2 to send out. The object then immediately sends out two more events: job J1 to the processor for the second half of its execution and message M2 to the LCN for delivery to its destination object. Note, in this scenario M2 is an asynchronous message as job J1 continues on with execution rather than waiting for a response to M2 before continuing execution. Again, the software object waits for an external event. On receipt of the completion of the second half of J1, the software object again selects an interaction arc A2 to create message M3, which is an asynchronous message. As job J1 has completely executed, the object creates the return message to the M1 invocation; and, as the object has no outstanding jobs, the object triggers the processor with an event to unload memory.

More complex scenarios are described in the following sub-sections, which depict the behavioral differences that can be encountered based on the setting of the software object thread mode. To help highlight some of these behavioral differences, some common configuration assumptions about the object and its environment are summarized here:

1) the object has two operation methods defined: O1 and O2;
2) the object has two arc interactions defined: A1 and A2, where A1 and A2 are asynchronous arcs;
3) the execution of operation O1 fires arc A1; O2 fires A2;
4) the processor for this object is multi-tasking and no other software objects are interacting with it;
5) the object initially receives three messages: M1, M2, and M3; only message M1 requires a response; M1 and M2 trigger selection of method O1 for execution while M3 selects O2.

The Parallel DEVS with ports specification for the behaviors of the swObject model is provided in Appendix C.

5.2.1. swObject Dynamics For Thread Mode None

In the "none" level thread mode, a *swObject* only executes one job at a time and any additional requests are queued. So this scenario, as shown in Figure 24, results in the sequential execution of jobs J1, J2, and J3. Job J1 is the initial active job, with jobs J2 and J3 being queued.

At the completion of J1, the M1 return message is sent as well as the new message M4, which was created based on the definition of interaction arc A1 and the completion of method O1. Similarly, with the completion of job J2, message M5 is created from A1 and sent. Finally, with the completion of J3, message M6 is sent based on arc definition A2, and the software object sends an event to "unload" memory.

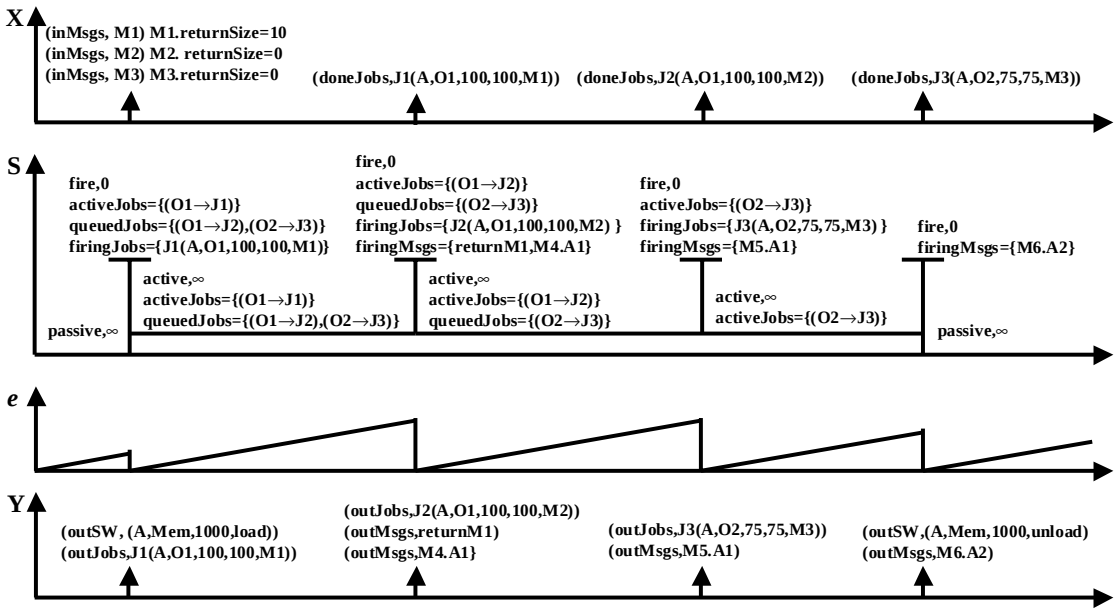


FIGURE 24. DCO swObject Discrete Event Time Segments: Thread Mode None

5.2.2. swObject Dynamics For Thread Mode Object

In the "object" level thread mode, a *swObject* can have one job per defined method concurrently active. So initially, our *swObject* has two active jobs: J1 associated with method O1 and J3 associated with method O2. As shown in Figure 25, job J2 is queued until completion of method O1 via the execution of J1.

As job J3 has the smaller work load and our processor is multi-tasking, J3 completes first and is the next external event. As J1 is still executing, J2 remains in queue, but the completion of method O2 triggers the sending of arc A2 with the creation of message M6.

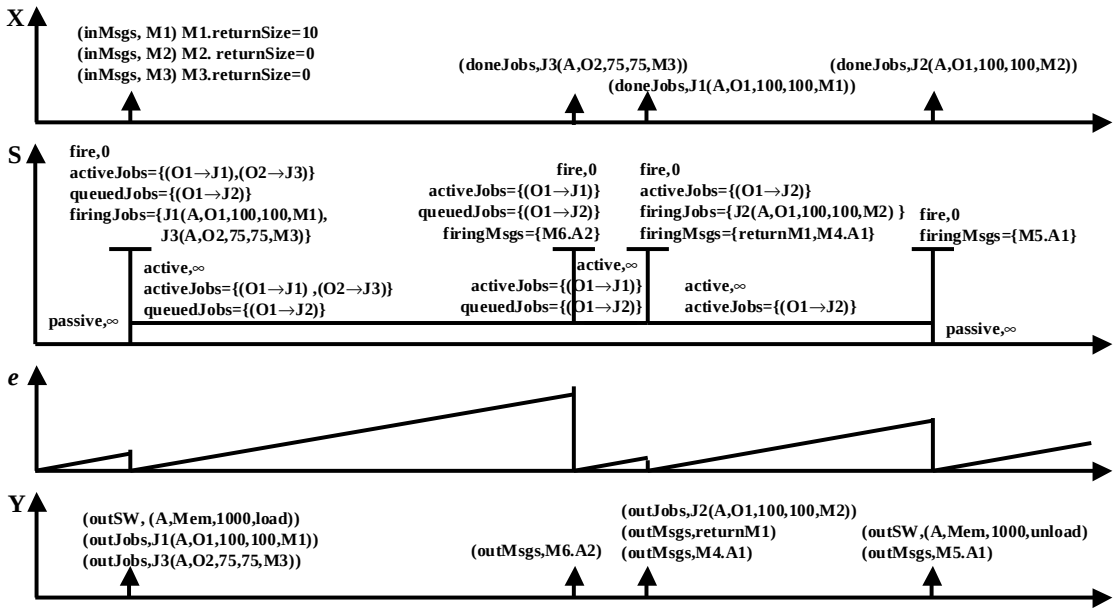


FIGURE 25. DCO swObject Discrete Event Time Segments: Thread Mode Object

With the completion of job J1, J2 becomes the remaining active job. The M1 response message is sent along with the message M4 based on method O1 being executed. Completion of job J2 closes out this scenario with the release of message M5 and the "unload" memory event. In contrast to the "none" thread mode event scenario, the queuing of jobs in the "object" thread mode scenario imparts prominent behavioral differences that are manifest in the relative timing and sequencing of output jobs and messages.

5.2.3. swObject Dynamics For Thread Mode Method

In the "method" multi-threading mode, a *swObject* reacts to all incoming requests with the generation of active jobs. Hence, a *swObject* in "method" mode starts out with the production of three active jobs: J1, J2, and J3. With J3 having the smallest work load

of the three, and since the processor is multi-tasking, the completion of J3 is our next external event as shown in Figure 26. The completion of J3 drops it from the active job list and also creates the M6 interaction message for output.

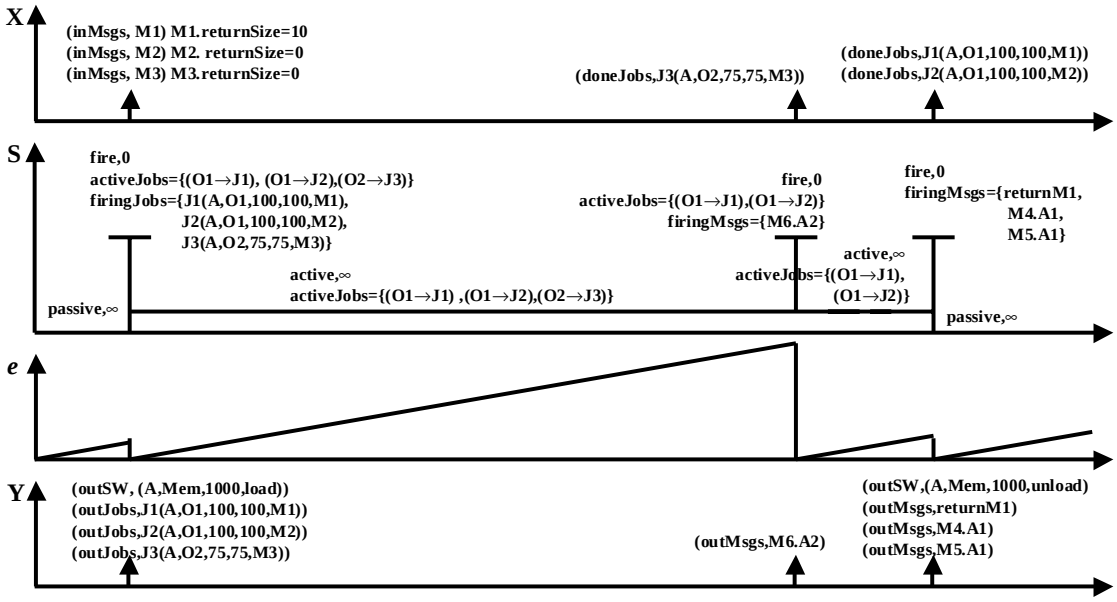


FIGURE 26. DCO swObject Discrete Event Time Segments: Thread Mode Method

The next external event is the completion of jobs J1 and J2. This clears out the active job list and creates three outgoing messages. The completion of J1 creates the return message for M1 and the new message M4 for executing method O1. Likewise, the execution of method O1 via J2 creates message M5. With no more active jobs, the *swObject* also sends an event to "unload" memory.

In contrast to both the "none" and the "object" level multi-threading modes, we observe another distinctly different behavior scenario for the same given input. The resulting job execution sequence and relative timings between the output messages have changed.

5.3. The OSM Component

The OSM component of the DEVS-DOC framework is the mapping of DCO software objects onto LCN processing nodes. In this mapping role, the OSM provides structural knowledge about the combined hardware and software system. The OSM, however, does not introduce any new behavior. Rather, the structures defined within the OSM impose constraints and limitations on the behavior of the individual components forming the DOC system.

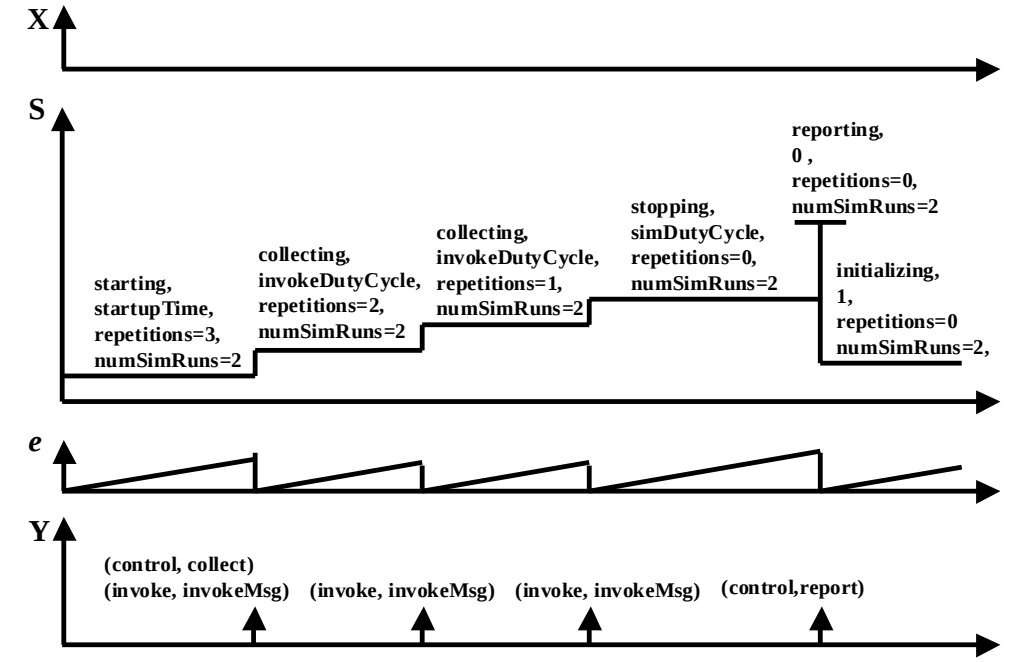
5.4. Experimental Frame Component Behavior

The dynamic behaviors developed for DEVS-DOC experimental frame components are summarized in this section.

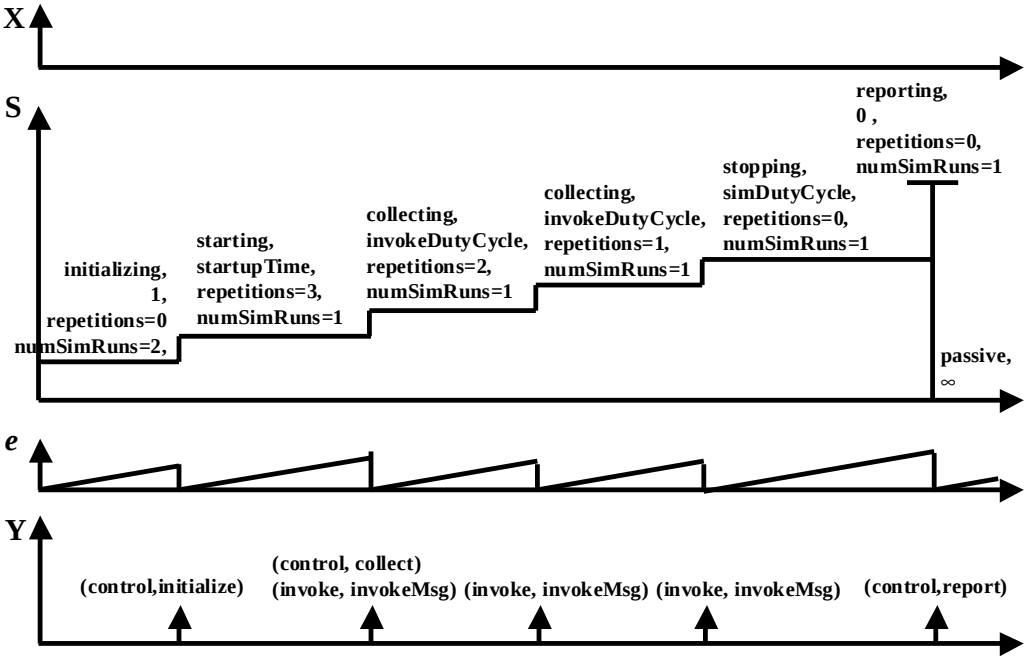
5.4.1. Acceptor

In an experimental frame, an *acceptor* monitors a simulation experiment to see that desired conditions are met. In the DEVS-DOC case studies described in chapter 6, we developed an *acceptor* to provide synchronization and coordination functions during the simulation experiments. The phases this acceptor cycled through are shown in the discrete event segment plots of Figure 27 a and b.

The time segment scenario depicted in Figure 27 is for the *acceptor* controlling two simulation runs of the same experiment (numSimRuns=2). The first plot depicts the time segments for *acceptor* control in running the first simulation experiment. The second plot, b, continues with showing the control of the second simulation experiment.



a. First Simulation Run



b. Second Simulation Run

FIGURE 27. Acceptor Discrete Event Time Segments

The *acceptor* starts out in a "starting" phase, which allows the DCO software objects time to send a load disk event to their respective processors. The *router* component of the processor model uses this event to begin configuring routing tables and forwards this routing information around to the other LCN components. The *acceptor* waits for these routing table configurations to complete based on the startupTime parameter.

The *acceptor* next signals all transducers to start collecting data by sending the "collect" event out the "control" port. The *acceptor* also stimulates the DEVS-DOC system model by sending an initial invocation message out the "invoke" port. For three repetitions, the acceptor continues to periodically — based on the invokeDutyCycle parameter — stimulate the system with invocation messages. The *acceptor* subsequently waits, for a period set by the simDutyCycle, for this first simulation run to complete. The *acceptor* then signals the transducers with a "report" event on the "control" port. The *acceptor* completes this first simulation control cycle by entering an "initializing" phase in preparation for controlling the second simulation run.

As shown in plot b, control of the second simulation run starts with the *acceptor* signaling all the DEVS atomic models to initialize themselves. The *acceptor* then cycles through the same sequence just described, with the exception of the *acceptor* ending the cycle by entering a passive state, rather than an initializing state, at the end.

A Parallel DEVS with ports specification of the behavior of this *acceptor* is included in Appendix C.

5.4.2. LCN and DCO Control Instrumentation

The behavior of the just described *acceptor* has control interactions with the atomic models of the LCN and DCO components. In order for these control interactions to have the intended effect, the LCN and DCO components must be capable of receiving and reacting to these control events. The following DEVS specification, which is also listed in Appendix C, is incorporated into the LCN and DCO atomic models to provide this experimental frame control instrumentation.

$DEVS_{LCN\_and\_DCO\_Control} = < X, Y, S, \delta_{int}, \delta_{ext}, \delta_{conf}, \lambda, ta >$, where

InPorts = { control }

X = { (control, controlMsg) }, where controlMsg = {passivate, initialize}

$S = Phase \times \sigma$

Phase = defined in LCN or DCO model

$\sigma = \Re_0^+$

$\delta_{ext}(s,e,(InPorts,X)) = (ph,sigma)$ for each x event on "control"

where ph is new Phase, and

sigma is new σ,

if x=passivate	ph=passive
	sigma=∞
else if x=initialize	initialize_LCN_or_DCO_model

$\delta_{int}(s) = (s)$

$\delta_{conf}(s,ta(s),x) = \delta_{ext}(\delta_{int}(s),0,x)$

$\lambda(s) = \varnothing$

$ta(s) = \sigma$

5.4.3. Transducer

The *transducer*, in an experimental frame, observes and analyzes the outputs of the system model under investigation. As previously described, our DEVS-DOC transducers need to respond to both control events from the *acceptor* and to statistical

events from the LCN and DCO components under observation. So, we develop a generic *transducer* behavior model to respond to the *acceptor* and then extend the generic *transducer* model to construct specific transducers to collect data for the metrics outlined in Table 1 of Chapter 4.

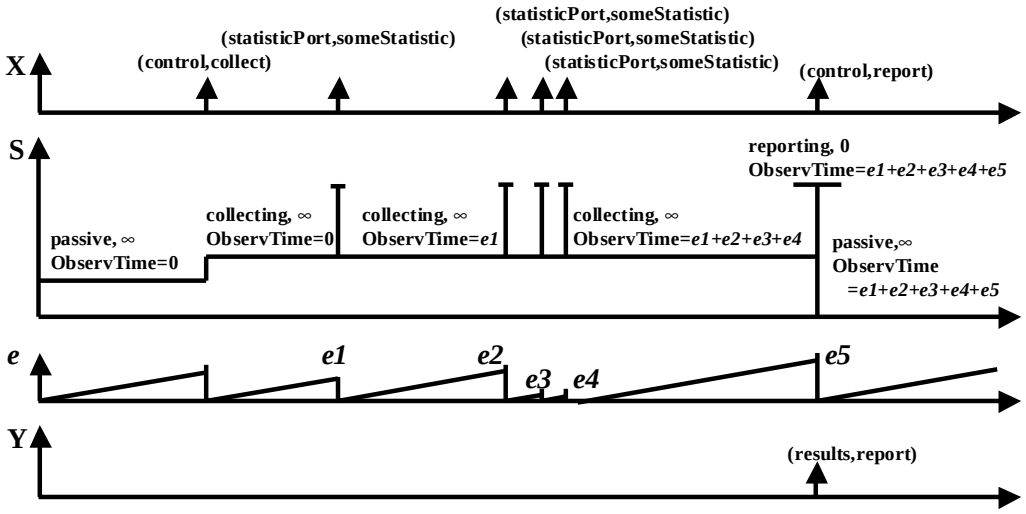


FIGURE 28. Transducer Discrete Event Time Segments

As illustrated in Figure 28, the generic *transducer* receives control events on its "control" input port. With receipt of a "collect" control event, the transducer begins collecting statistic events on specific ports. While in this collecting phase, the transducer keeps track of the observation interval with the "ObservationTime" state variable. On receipt of a "report" control event, the transducer generates a report on its observations and passivates.

A Parallel DEVS with ports specification of this behavior is included in Appendix C. The behavior of specific LCN and DCO transducers to collect the metrics outlined in Table 1 simply extend this generic behavior by defining the statistics to collect, the input

ports to observe them on, and a generate report function for the generic *transducer* to call as part of its output function.

5.5. Synopsis

Each component in the DOC systems model represents a real world entity. These DOC components characterize key structural and functional attributes associated with the real world counterparts. This chapter details a formal behavioral specification for each component in our DEVS-DOC environment. These behavior specifications define the dynamic manner in which these components act and react.

For each component, its structural composition is defined in terms of input ports, output ports, and a set of sequential states. The behavioral specification defines the set of input values allowed on each input port; the set of output values that can be sent on each output port; the state transition functions (internal, external, and confluent) which define the state sequences; the output function; and the time advance function. Each of these aspects is formally specified in DEVS. Fragments of these specifications are discussed in this chapter with complete specification listings provided in Appendix C. To supplement these component behavior discussions, discrete event trajectory diagrams depict how various event scenarios affect component state transitions and outputs.

6. CASE STUDIES

This chapter of the dissertation presents four case studies that use DEVS-DOC to model various distributed computing applications. The four studies demonstrate the breadth and depth of applicability for the DEVS-DOC environment. The first case study is of a network management application monitoring the status of managed devices across a local area network [Hil99]. This study was not only modeled and simulated in the DEVS-DOC environment, but was similarly configured and run in a real world environment allowing for a comparison of the simulation results with real world behavior. The second case study is of an HLA-compliant distributed simulation federation [Zei99c] that also has a real world counterpart to allow us to compare simulation results against. The third study is a model examining the interactions of an email application involving an email server, a name server, and email clients [Hil98b]. Wherein the first two cases we used *directed* modeling to specify specific software object method and arc interaction sequences, this study uses the *quantum* modeling approach. In this study, we also present and discuss several of the more interesting simulation results collected and plotted using the DEVS-DOC environment. The final case study revisits the email application case study with a focus on LCN alternatives. This case study demonstrates the degree of independence achieved in modeling the DCO, LCN, and OSM components while also demonstrating the degree of interdependence on overall system behavior.

6.1. Simple Network Management Protocol (SNMP) Monitoring

In this DEVS-DOC case study, we have modeled and simulated an Internet Engineering Task Force compliant SNMP management system [IET90]. Such a system has four essential elements: management stations, management agents, management information bases (MIBs), and a management protocol. Management stations, using the management protocol, request management agents to perform management operations on MIB objects, and the agents respond to these requests. MIB objects represent manageable attributes. For our scenario under study, the manager requests agents to provide status on selected MIB objects via the *snmpget* command.

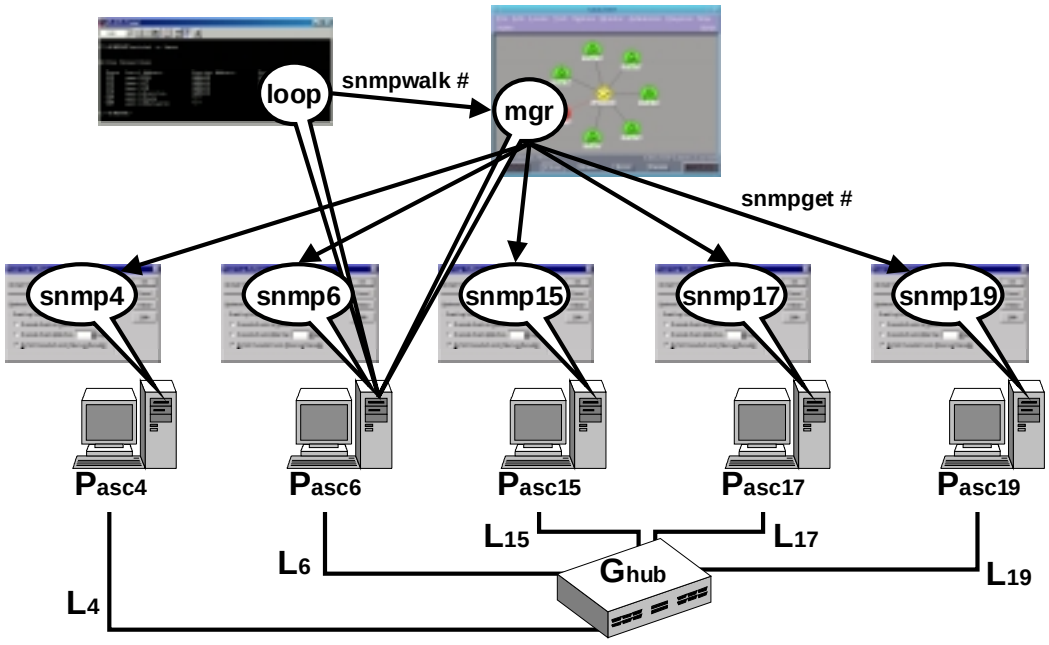


FIGURE 29. SNMP Monitoring

The DOC system under study is depicted in Figure 29. The LCN consists of five host processors interconnected with 10 Mbps ethernet links through a central hub to form

a network with a star topology. For the DCO, each processor has an SNMP agent. One of the five processors (Pasc6) also has an SNMP manager software object (mgr) and a *loop* controller software object (loop). For the experimental frame, a single computational domain is defined to encompass all five *SNMP* agents, the *manager* and the *loop* software objects.

The *loop* controller drives the overall system dynamics for this scenario. The *loop* object fires an *snmpwalk* interaction arc to invoke the *manager* object to “walk” the MIB of one of the five host processors. With this invocation, the *manager* fires a series of *snmpget* commands. These commands translate into *snmpget* jobs that load the processor's cpu and then fire *snmpget* interaction arcs that query (invoke) the targeted *SNMP* agent. The *SNMP* agent fires a job to process the request and then fires a return arc representing the response to the query.

The system entity structure for this case study is depicted in Figure 30. The system is decomposed into the LCN components consisting of processors, media access units (MAUs), links, and gates; the DCO swObjects; and the Experimental Frame components consisting of a domain transducer and an acceptor. Key couplings for this decomposition are annotated directly under the "SNMP System-aspect." For instance, coupling of the LCN is summarized in the five lines listing the coupling of a processor to its MAU, to its link, to a numbered port on the hub. Attribute settings used within this case study are also annotated next to the appropriate entities in the SES. The DEVS-DOC code to model and simulate this system is provided in Appendix B.

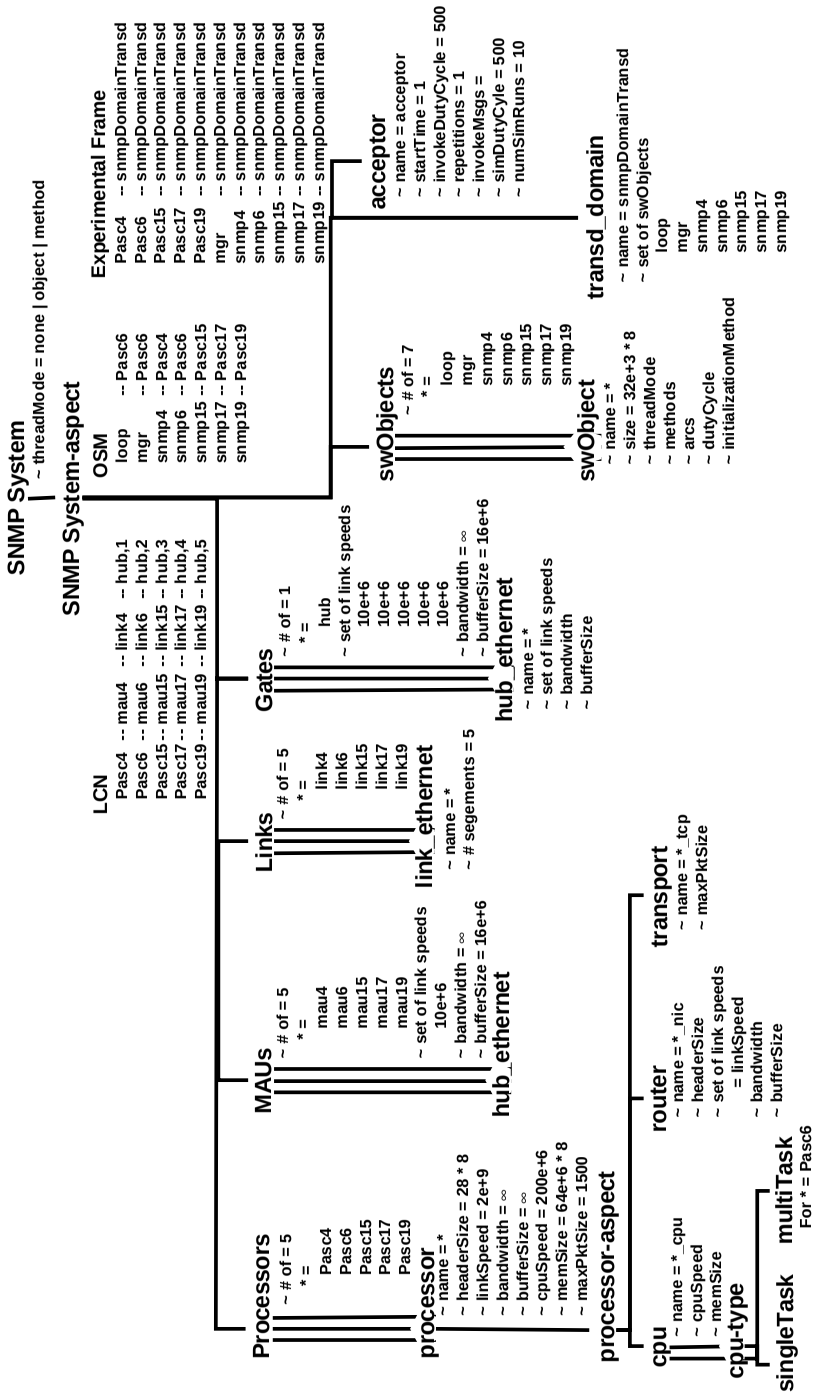


FIGURE 30. SES for the SNMP Monitoring Case Study[6]

[6] Infinity (∞) is used as a model simplification setting for several component parameters in this case study. In the DEVSJAVA implementation, INFINITY is defined as a double sized real number with a static, final setting of POSITIVE_INFINITY.

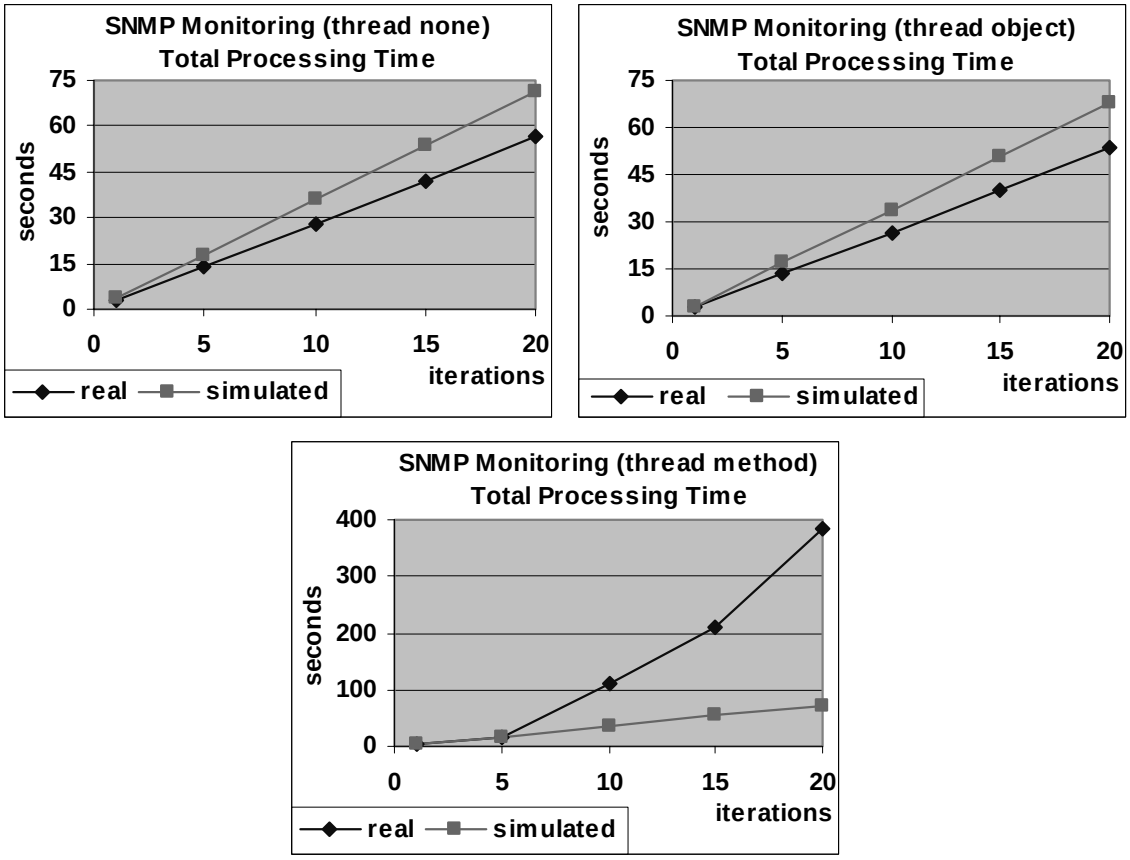


FIGURE 31. SNMP Monitoring System Total Processing Time

During the investigation, we ran three series of simulations. For each series, we set the *manager* software object to *none*, *object*, and *method* level multi-threading on its thread mode. Within each series, we executed ten simulations with the *loop* software object set to fire the *snmpwalk* arc the same number of times as the simulation iteration. In other words, fire *snmpwalk* once for the first simulation; fire *snmpwalk* five times for the second simulation; ten times for the third simulation; 15 times in the fourth simulation; and 20 times in the fifth simulation. So, the iteration count represents an aggregate workload. We also realized this DOC system in a lab environment and

collected data on the execution times. Figure 31 depicts plots of the total processing time results for the simulation runs (simulated) and the real system measurements (real), where the iteration count represents the aggregate workload to be processed.

From these results we see that the total processing time begins to rapidly increase for the real system when the *manager* is set for a *method* level of granularity. We explored this behavior further using the system activity reporter (sar) utility on the Unix system running the *manager* object. From the sar report, we found that after ten iterations significant amounts of time accumulated for block transfers into and out of memory. Our real system *manager* configuration is such that it actually starts a new process for each *snmpget* method, and with the *method* level granularity, ten loop iterations, and 35 *snmpgets* per iteration, it equates to 350 concurrent execution requests. The block transfers end up swapping out the processes executing these requests.

Our simulation results for this case study correspond well with most runs. For the *method* level configuration, however, our poor results are strongly attributable to a weak representation of memory swapping dynamics within our DEVS-DOC *cpu* model. Developing an improved *cpu* memory representation is one instance of where an improved representation of information technologies is needed within DEVS-DOC. This need is discussed further in the information technologies sub-section in the final chapter of this dissertation.

6.2. Distributed Federation Simulation

This case study is of a DEVS/HLA[7]-compliant distributed simulation federation [Zei99c] that also has a real world counterpart to allow us to compare simulation results against. In [Zei99a] a pursuer-evader federation is implemented in DEVS/HLA to investigate predictive contract mechanisms. Now, the same pursuer-evader federation is studied to show applicability of the DEVS-DOC environment and to demonstrate DEVS-DOC as a means of exploring the tradeoffs in message traffic and computational loads in a complex distributed computing environment.

6.2.1. Predictive Contracts and DEVS/HLA

DEVS/HLA is an HLA-compliant modeling and simulation environment formed by mapping the DEVS C++ system [Zei97b] to the C++ version of the DMSO RTI [DMS98a and DMS98b]. While HLA supports interoperation at the simulation level, DEVS/HLA supports the hierarchical and modular modeling construction features inherited from DEVS.

The operational form of the HLA is a Run Time Infrastructure (RTI) that supports communication among simulations, called federates. DMSO (Defense Modeling and Simulation Office) has developed an RTI in C++ for use in the public domain [DMS98a]. HLA supports a number of features including establishing, joining, and quitting federations, time management, and inter-federate communication [Dah98].

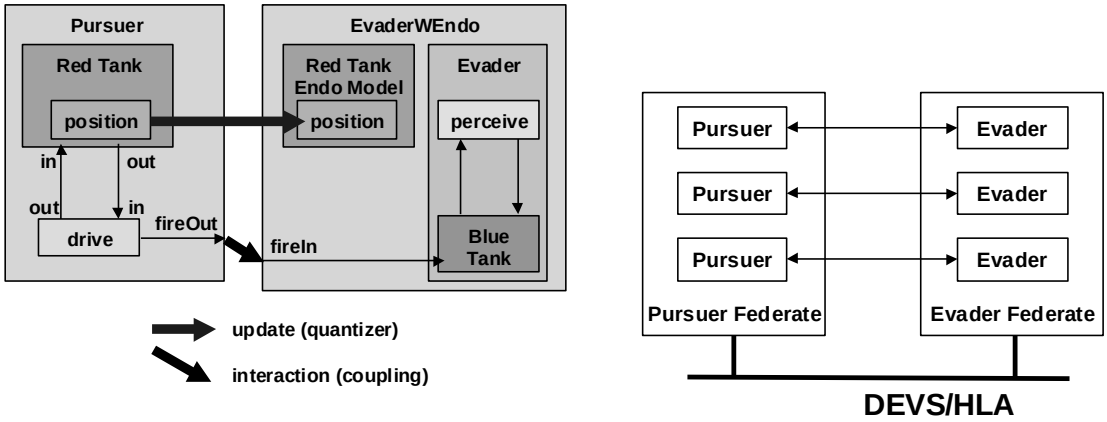
[7] HLA is the DoD Modeling and Simulation Office (DMSO) High Level Architecture for implementing distributed simulations [Dah98], [DMS98a], [DMS98b], and [DMS99].

Predictive contract mechanisms are an active area of research into technologies to support large distributed simulations [Cha79, Cho99, Zei98, Zei99a, and Zei99c]. The idea behind predictive contract mechanisms is to reduce the message traffic exchanged between distributed simulation federates while incurring only marginal additional computational overhead. In [Zei99a, Zei99c, and Cho99], tradeoffs between computational overhead and bandwidth requirements for various predictive contract mechanisms were examined using the concept of quantization [Zei98].

In [Zei99a], three predictive contract mechanisms were studied using a pursuer-evader model to illustrate performance impacts within the simulations. The first mechanism was called *non-predictive* quantization, wherein a sender federate updates the receiving federate with a numerical, real-valued, state variable, each time an agreed upon (contracted) threshold is crossed. The second mechanism was called *predictive* quantization. In predictive quantization, only a one-bit message is sent when a contracted threshold is crossed; the one-bit message signals a crossing of the next higher or next lower boundary. The third mechanism was called *multiplexed predictive* quantization, which expanded on the predictive quantization concept. As mentioned above, predictive quantization reduces the information sent about boundary crossings to a single bit. Given the overhead bits associated with creating a packet to send this information, reducing the payload from 64 bits to 1 may not produce a significant overall reduction. However, when large numbers of interacting entities reside on each federate, it is possible to multiplex their reduced outputs into a single packet and exploit the message size benefits

of predictive quantization. This concept defined the *multiplexed predictive* quantization mechanism.

The pursuer-evader federation depicted in Figure 32a was used in [Zei99a] to explore these predictive contract mechanisms. The pursuer and evader components were implemented as DEVS coupled models of vehicles and drivers that follow simple rules for pursuit and evasion. The Pursuer component contains a red tank model that reflects its state to the EvaderWEndo component. The EvaderWEndo component contains an endo-model [Zei90] of the red tank, which receives the reflected state values from the Pursuer. The EvaderWEndo component also contains an Evader model, which itself contains a blue tank model. A simple federation, Figure 32b, is formed of two federates – a Pursuer federate and an Evader federate. Matched pursuer - evader pairs create dynamics for federate interactions during simulations.



a. Pursuer-Evader Pairs **b. Pursuer-Evader Federation**

FIGURE 32. Pursuer – Evader Federation

In studying the effects of the two extreme quantization mechanisms within a distributed simulation setting, a DEVS/HLA model with the two federates was configured to hold an arbitrary number of matched pursuer-evader pairs. An experiment consisted of a number of randomly initialized identical pairs being simulated for a set simulation time (100 cycles) and quantum size. In successive experiments, the number of pairs was increased to approach network saturation.

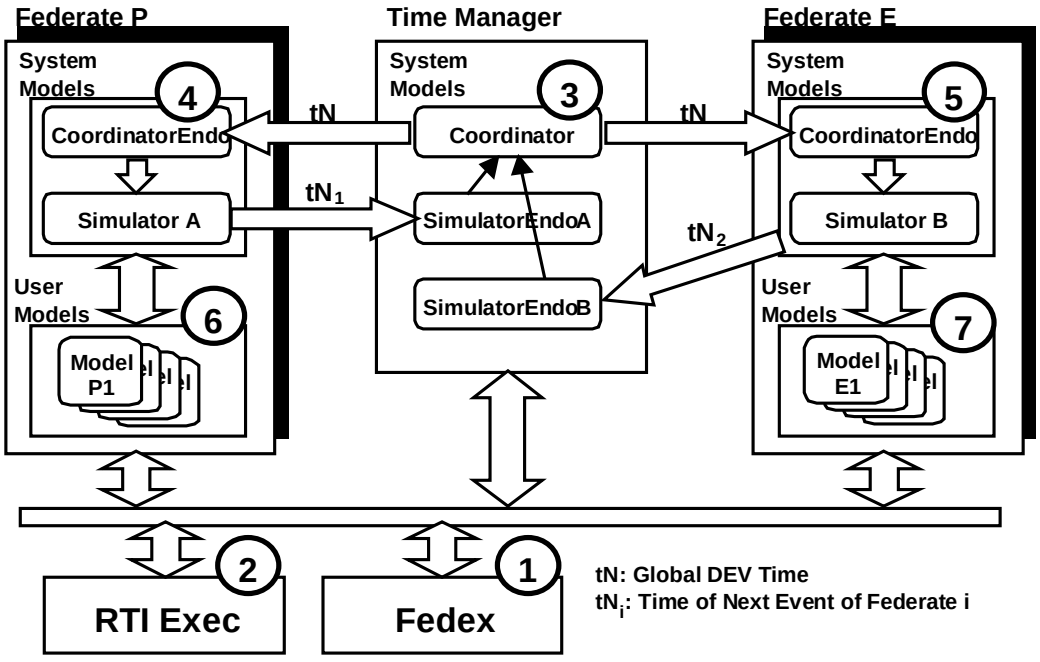


FIGURE 33. DEVS/HLA Federation Infrastructure

In implementing the federation, each federate was configured on a Unix workstation with the two workstations being interconnected via an Ethernet link. Figure 33 depicts the DEVS/HLA federation infrastructure used to implement the simulation federation. Federate P and E are formed to support the Pursuer and Evader federates. The time manager federate is formed to enable the DEVS coordinator as described in

[Zei99b]. The RTIexec and Fedex are DMSO HLA standard components [DMS98a DMS98b].

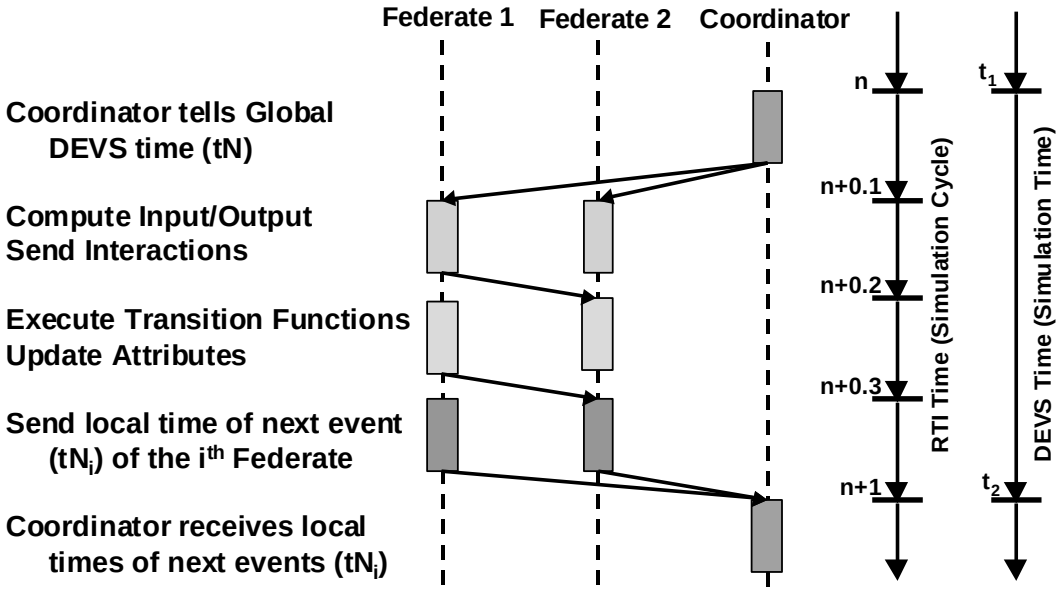


FIGURE 34. Combined DEVS/HLA Simulation Cycles

Due to a constraint imposed by the HLA RTI rules, see [Zei99b], the RTI Time Management mechanisms are used to synchronize the DEVS Simulation Cycle as illustrated in Figure 34. Logical time (DEVS time) proceeds along a separate axis, as shown, and advances each cycle in the phase where the coordinator updates each federate with the DEVS global time.

The results of the study reported in [Zei99a] are summarized in Figure 35. As the number of pairs increased, the simulation execution time increased in a highly non-linearly fashion for the *non-predictive* quantization mechanisms. In contrast, simulation execution time for the *multiplexed predictive* quantization mechanism demonstrated a

significant performance improvement with only a very marginal increase in execution times for 1000 pairs or less.

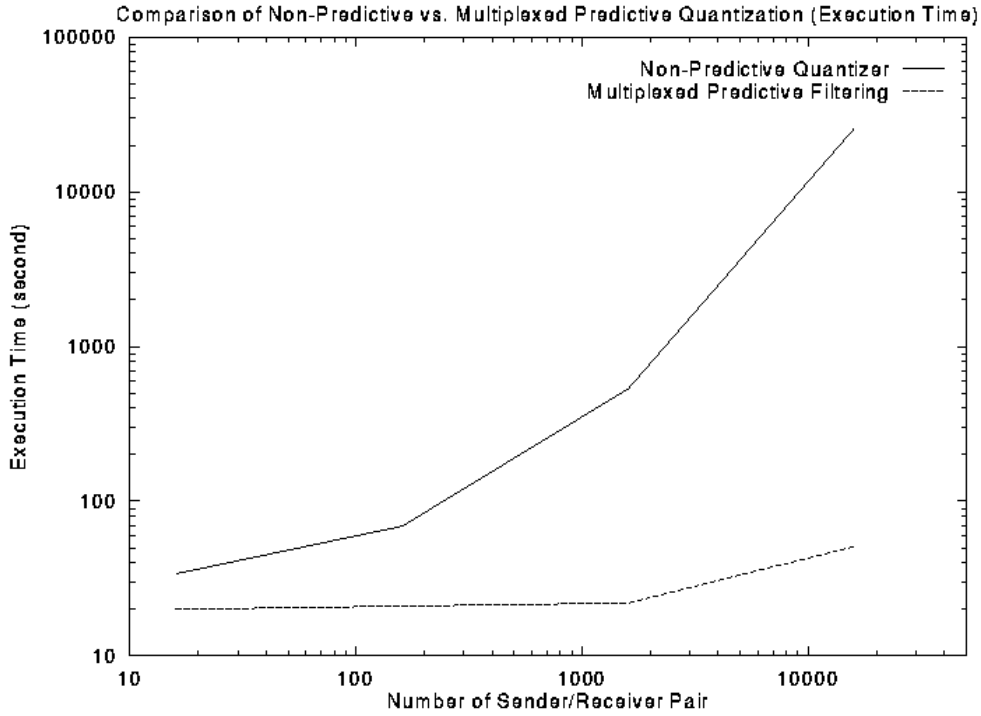


FIGURE 35. Combined DEVS/HLA Simulation Cycles

6.2.2. Predictive Contracts and DEVS-DOC

We have employed the DEVS-DOC environment to model and simulate this distributed federation simulation. For our DEVS-DOC case study, the LCN simply consists of the two processors using media access units (MAUs) to interconnect with the ethernet link. For the DCO, seven software object models were defined as numbered in Figure 33; one each for the Fedex, RTI Exec, DEVS coordinator (time manager), Federate P coordinator endo-model, and the Federate E coordinator endo-model. For the set of pursuer models in Federate P, a single software object representing an aggregation

of the pursuers is defined. Similarly, for the set of evader models in Federate E, a single aggregate object is defined. The OSM places the Federate E coordinator and the evader aggregate software objects on one processor and all the other DCO objects on the other processor. The Experimental Frame consists of two transducers and an acceptor. A software object transducer collects data on the RTI Exec, and an ethernet link transducer monitors the network link. An acceptor monitors and controls the simulation runs.

As the number of pursuer-evader pairs increases, the performance of the simulation under each predictive contract mechanism is a central focus of the experiments. Defining the set of pursuer models and the set of evader models as aggregate software objects provides a simpler means of scaling the complete DEVS-DOC system model. The computational workloads for the pursuer aggregate DCO object is a function of the number of objects being represented. Similarly, the communications traffic load placed on the LCN by the DCO is scaled by increasing message sizes based on the number of pursuer-evader pairs.

For experiments, we ran 100 DEVS simulator cycles. During each cycle, we assumed that 50% of the pursuer-evader pairs were active with interactions. To model the non-predictive quantization mechanism, we assumed the contracted threshold settings resulted in a five-fold decrease in message traffic, i.e., only 1 in 5 of the interacting pairs actually triggered a threshold crossing. The message size payload is based on the 64 bits needed to represent the numerical, real-valued, state variable. For the predictive quantization mechanism, the same five-fold decrease in traffic is assumed, but now the message size payload is one bit long. For the multiplexed predictive quantization

mechanism, a 1-bit payload is again assumed, however, these 1-bit payloads are multiplexed into a single message for routing through the RTI Exec.

Interactions for the DCO software objects were defined based on the abstract interactions depicted in Figure 34. To help elaborate these DCO software object interactions and associated method invocations, the UML interaction sequence diagram in Figure 36 was developed as a detailed refinement of Figure 34. Message sizes associated with the pursuer and evader interactions were set as outlined in [Zei99a]. The message sizes associated with the federate coordinators, Fedex, and RTI Exec objects were qualitatively estimated to reflect the complexity of information contained in these exchanges. Similarly, the computational workloads for the methods associated with each software object were qualitatively estimated to reflect the relative computational complexity of the functions represented.

Figure 37 depicts the system entity structure for the set of DEVS-DOC models developed under this case study. As in the SES diagram of the previous case study, the key couplings associated with this decomposition are annotated directly under the "Pursuer-Evader Federation-aspect," and attribute settings used within the case study are also annotated next to their associated entities. The DEVS-DOC code for this model is provided in Appendix B.

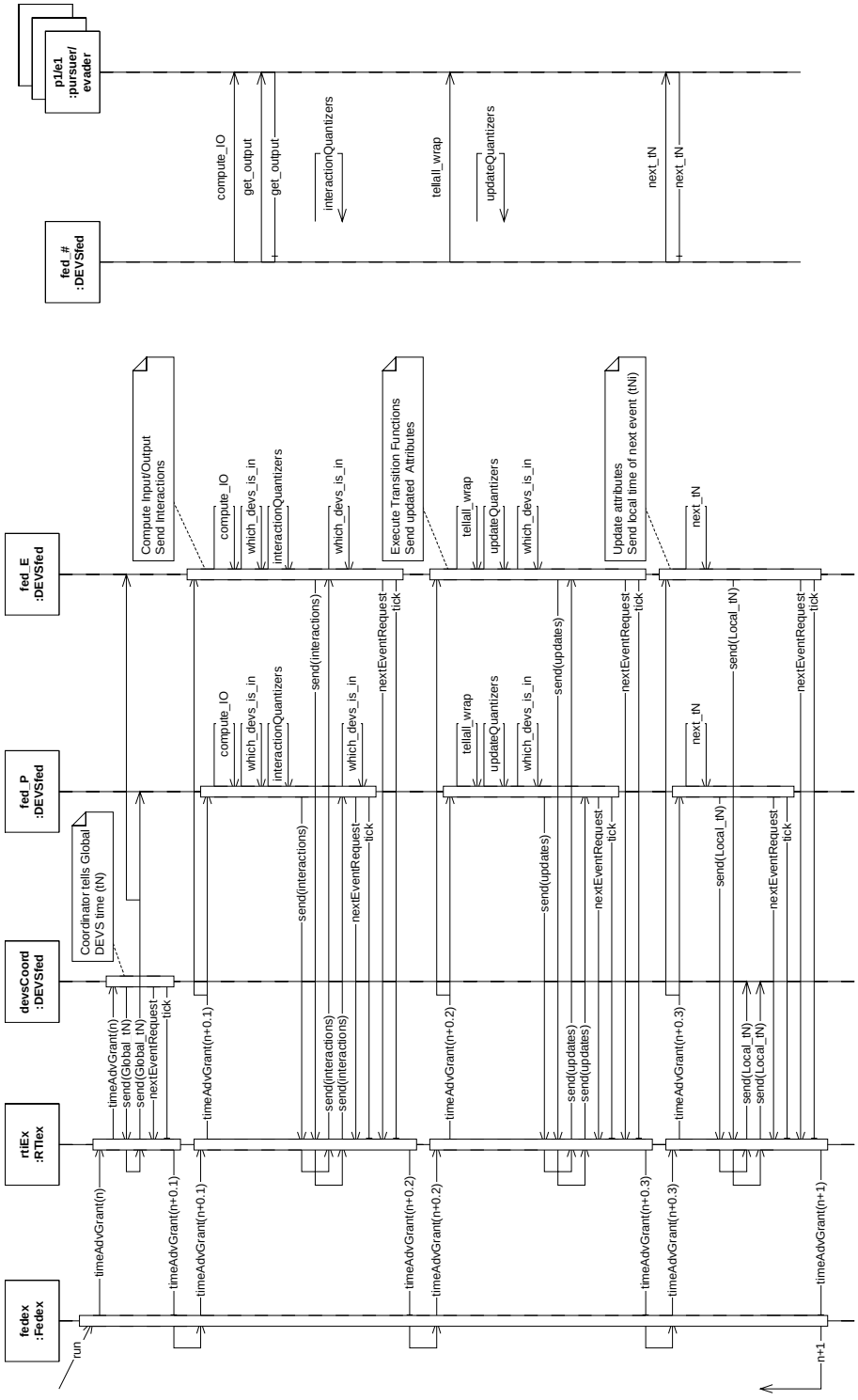


FIGURE 36. DEVS/HLA Pursuer - Evader Federation Interaction Sequence Diagram

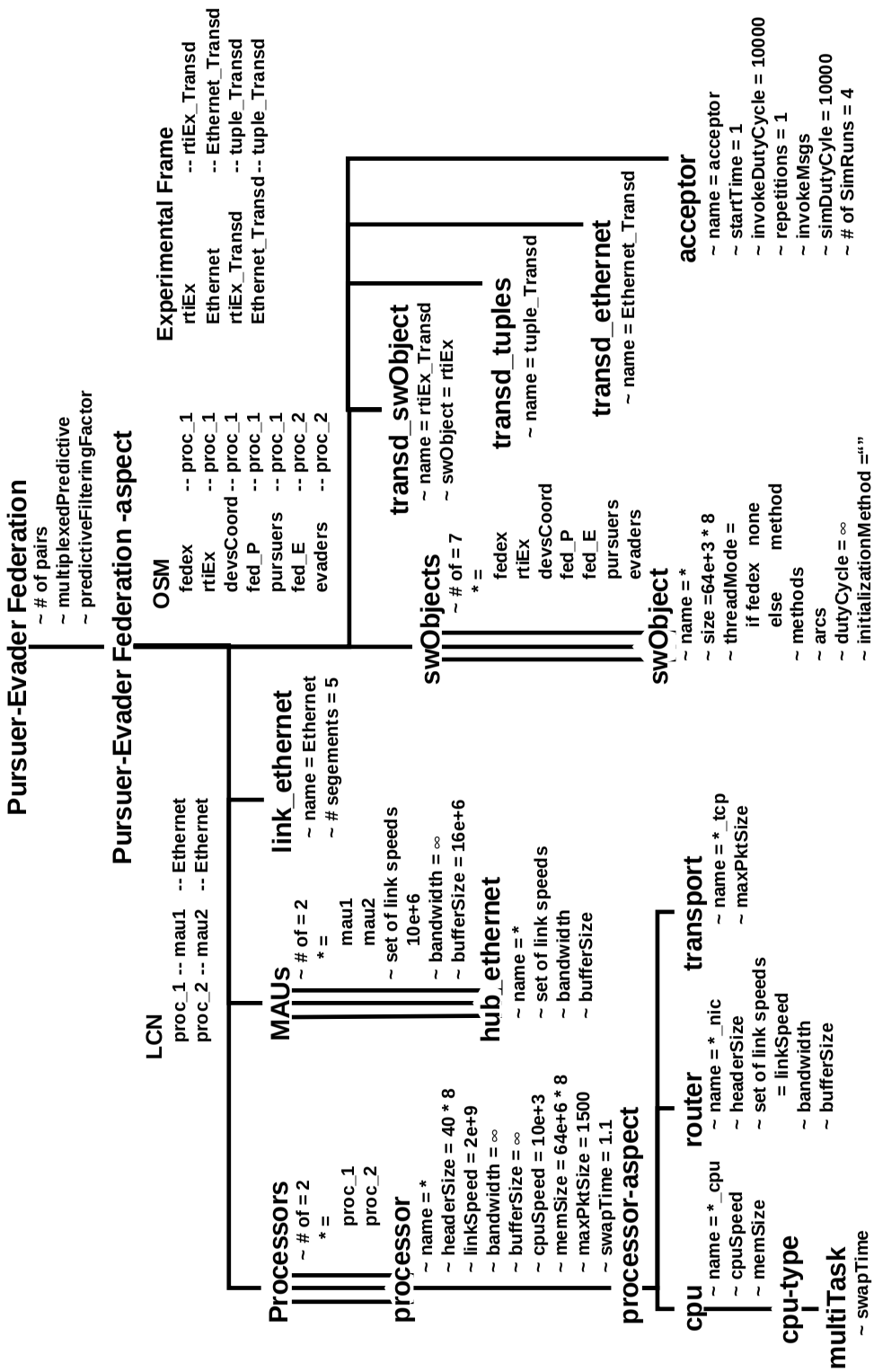


FIGURE 37. SES for the DEVS/HLA Pursuer - Evader Federation Case Study

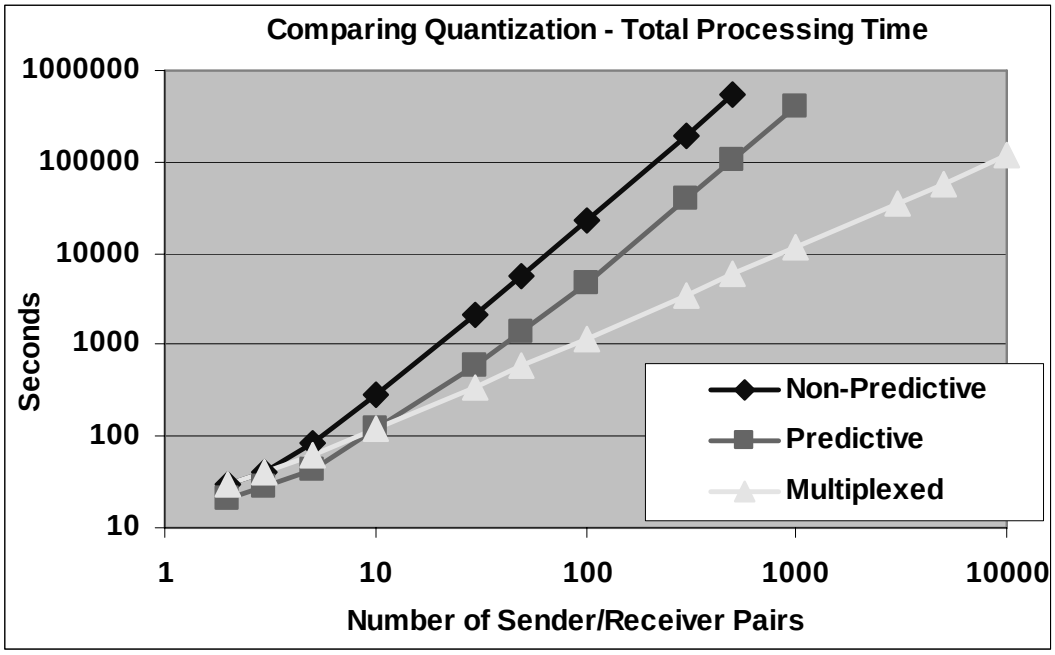


FIGURE 38. Federation Total Processing Time versus Number of Pursuer-Evader Pairs

Results from simulating this DEVS-DOC federation model are depicted in Figure 38. The absolute magnitudes associated with these results are not that interesting as they can be easily shifted by changing each processors' speed parameter or with adjustments to the computational workloads associated with the methods in each software object. The relative trajectories for each simulation series – non-predictive, predictive, and multiplexed predictive – are quite interesting in that they strongly reflect the behaviors found in the original study [Zei99a].

As will be shown, the simulated computational loads dominate the total processing time by more than an order of magnitude. Adjusting the modeled computational loads to better characterize the real system would result in stronger

network effects and a closer correlation of these simulated *total processing time* results with the real system measurements.

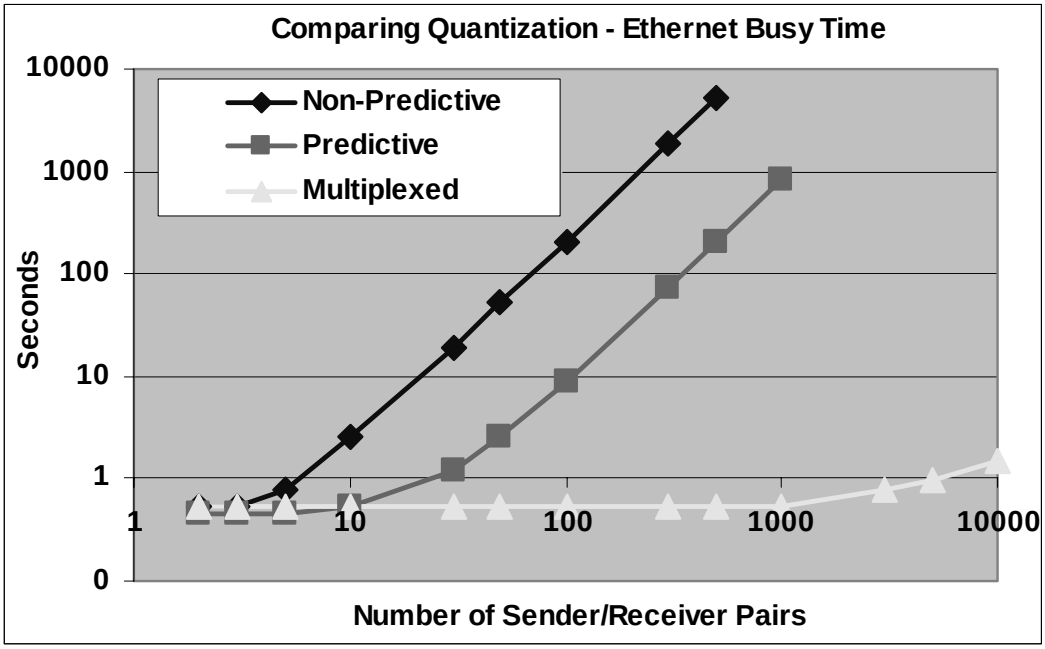


FIGURE 39. Ethernet Busy Time versus Number of Pursuer-Evader Pairs

During this DEVS-DOC investigation of the federation model, we also collected results on the cumulative time the ethernet link was busy with transmissions. These results are shown in Figure 39. Within DEVS-DOC, this "busy" time represents the time spent on successfully transmitting traffic as well as time spent transmitting and recovering from collisions. However, no collisions occurred during any of the simulations. This simulated busy time represents only a fraction of the simulated run time plotted in Figure 38. The simulated computational load dominates the simulated run time.

Observing no collisions is reasonable as only two workstations are modeled; no other traffic source is represented; the traffic load occurs in lock step with the simulated DEVS cycle; and the simulated computational load significantly dominates the overall simulated run time.

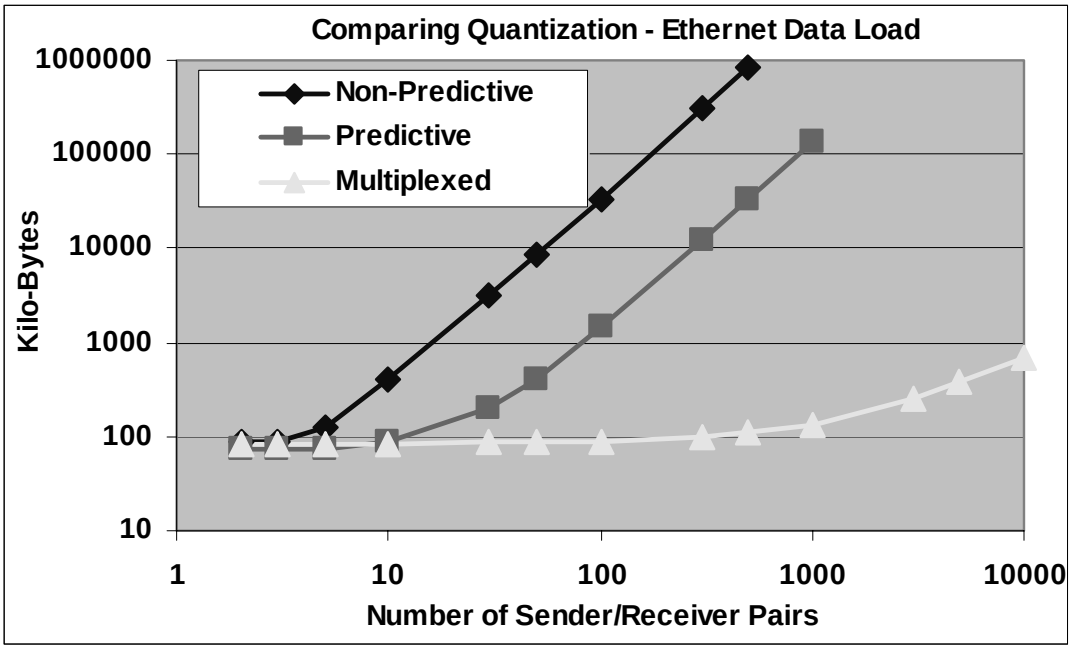


FIGURE 40. Ethernet Data Load versus Number of Pursuer-Evader Pairs

The simulated total data load transmitted across the ethernet is depicted in Figure 40. As no collisions occurred in any of the simulation scenarios, the data load transmitted is a direct complement to the previously discussed Ethernet Busy Time plot.

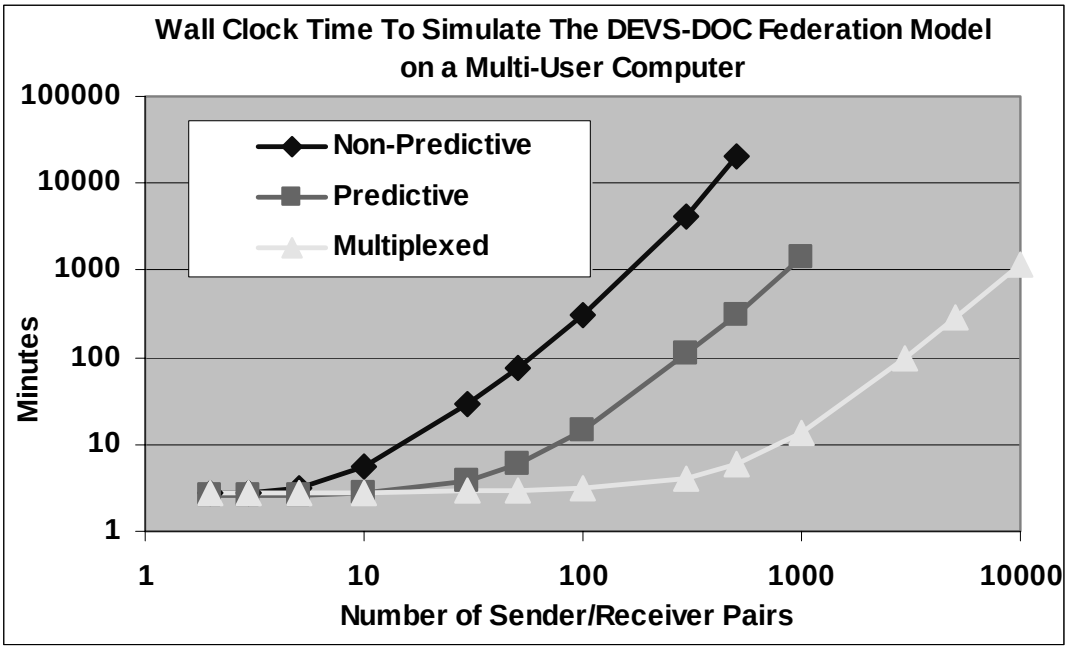


FIGURE 41. Simulation Execution Time For DEVS-DOC Federation Model

In Figure 41 we depict the real (wall clock) execution time to run these DEVS-DOC simulations of the distributed DEVS/HLA federation. For example, the Predictive simulation of 1000 sender/receiver pairs took just over 1000 minutes (approximately one day) to run. This one-day run actually is executing the scenario four times with the average of these results being used to generate the plots in figures 38 through 40. These simulation runs were executing on a 4 processor, 250 Mega-hertz, 1 Giga-byte, multi-user machine. Processing loads from other users during these simulation runs was not monitored.

6.3. Email Application

In this case study we consider an email application comprised of three specific components: an *Email Client*, an *Email Server*, and a *Name Server*. Within the Internet Engineering Task Force (IETF) architecture and standards, the Email Server represents a Simple Mail Transfer Protocol (SMTP) server, while the Name Server represents a Domain Name Service (DNS) server. Within the International Standards Organization (ISO) architecture and standards, the Email Server represents an X.400 server and the Name Server represents an X.500 server. In both cases, the Email Client represents an end user's client application.

The Email Client component is modeled as a DCO software object (swObject) that has a specification of two methods and two interaction arcs: an invocation arc and a message arc. The two methods represent a function to resolve email names to addresses within an email, and a function to send an email to the Email Server. The invocation arc represents a request to the Name Server to resolve names associated with an email. The message arc represents the sending of an email message via the Email Server.

Similar to the Email Client, the Email Server component is modeled as a DCO swObject with its own specific methods and arcs representing an electronic mail service. The Email Server has three methods, three message arcs, and one invocation arc. The three methods represent functions to receive email messages, forward email messages, and resolve addresses. The three message arcs represent three different email messages that are processed through the server. Two of these messages have Email Clients as

destinations. The invocation arc represents requests to the Name Server for resolving addresses.

The remaining software object for this Email Application is the Name Server component. Its DCO swObject model has specific methods and arcs for a name service object. The Name Server has a single method defined to look up names (resolve names). No invocation or message arcs are defined. Thus, incoming arcs invoke the lookup method, and on completion, the Name Server object simply fires return arcs for each invocation arc that activated the server.

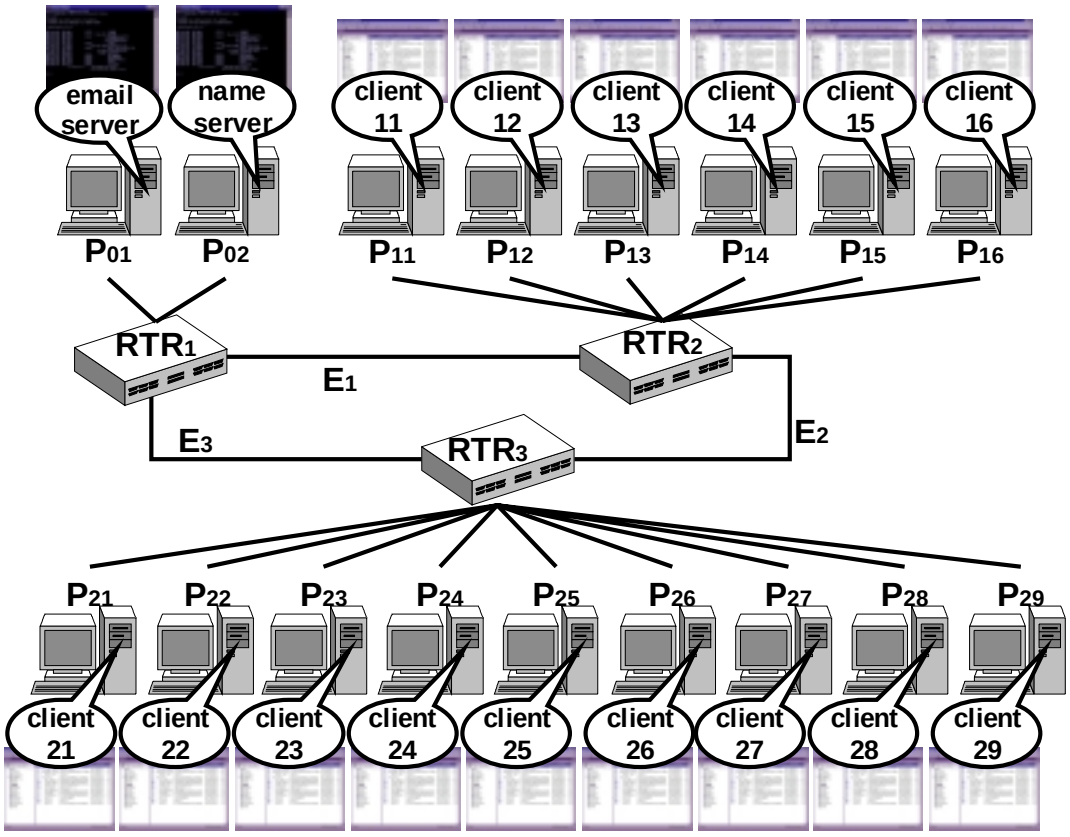


FIGURE 42. Email Application LCN Topology and OSM

For the LCN in this study, we explored a slightly more complex network. In particular, we modeled three routers, each interconnected with the other two. This configuration begins to test and demonstrate the routing logic within the LCN. The interconnection of the routers is via 10 megabits per second ethernet links. Off of each router is a set of two or more processors. Each processor has a dedicated link to its LCN router. This LCN structure forms a router-star topology as depicted in Figure 42. For the OSM, we mapped one software object onto each processor.

Each of the fifteen Email Clients exhibits the same behavior as expressed above. Each client is set with an *object* level multithreading mode. Each client is also configured to select the "resolve names" method 70% of the time it is invoked, and the "send email" method the remaining 30%. The Email Server is set to a *method* level of multithreading and selects the "receive email" method 20% of the time it is invoked, "forward email" 60%, and "resolve names" the remaining 20%. In investigating this system, two sets of simulations are run. One set has the Name Server set at no (*none*) multithreading and the other at *method*. As the Name Server only has one method, it is always selected when invoked.

For the experimental frame, an acceptor is coupled to each Email Client to invoke it forty times at the start of each of twenty simulation runs. Each run is allowed to progress until all the objects passivate. A software domain transducer for the email application is connected to each software object. Software object transducers are connected to the Email Server, Name Server, and one of the Email Client software objects. Ethernet transducers are connected to each ethernet. The transducers collect

system events from their components throughout each simulation run. The layered experimental frame concept described in section 4.5 is used to compute, from the transducer reports, the maximum, mean, and minimum statistical values across the twenty simulation runs for each of the two Name Server configurations.

The system entity structure for this Email Application case study is provided in Figure 43. This SES highlights the LCN, DCO, and Experimental Frame components and their couplings. The attributes settings used are also annotated.

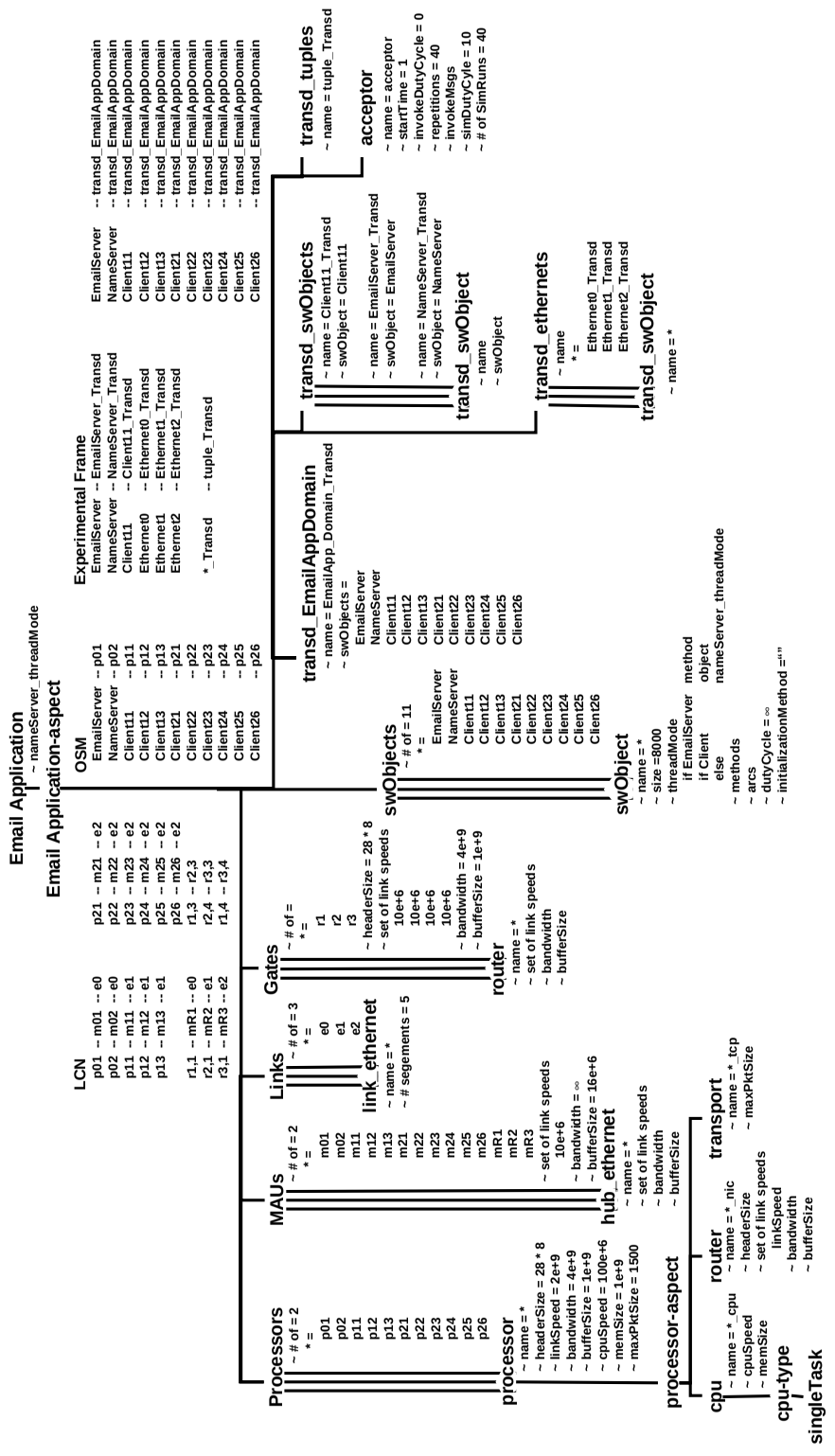


FIGURE 43. SES for the Email Application Case Study

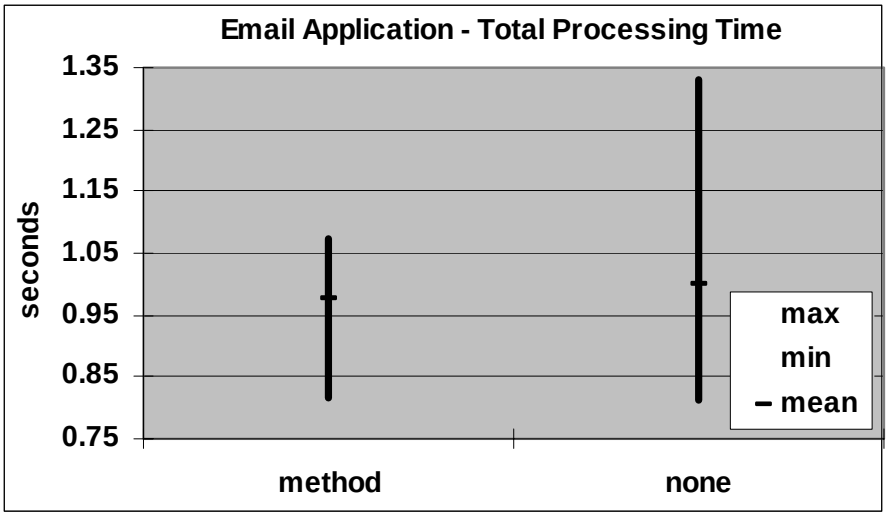


FIGURE 44. Email Application – Total Processing Time[8]

When the Name Server thread mode is set to *none*, only one resolve name request is processed at any point in time; subsequent requests get queued and processed on a first-in, first-out (FIFO) fashion creating a pipeline effect. We expect this pipelining to also impact the dependent processes (Email Clients and Email Server) and effectively reduce the amount of concurrent processing across the system. Thus, the "*none*" configuration should result in slightly longer run times than when the Name Server is configured for *method* level multithreading. Figure 44 plots the maxima, means, and minima collected from the twenty simulation runs for the two Name Server thread mode settings. While the simulation results do not demonstrate a significantly shorter total processing time for the *method* setting, they do show a smaller variation in the total time required to process the aggregate workload of the experiment.

[8] Note, the *none* and *method* labels in these plots reflect the thread mode setting of the Name Server object during the simulation runs.

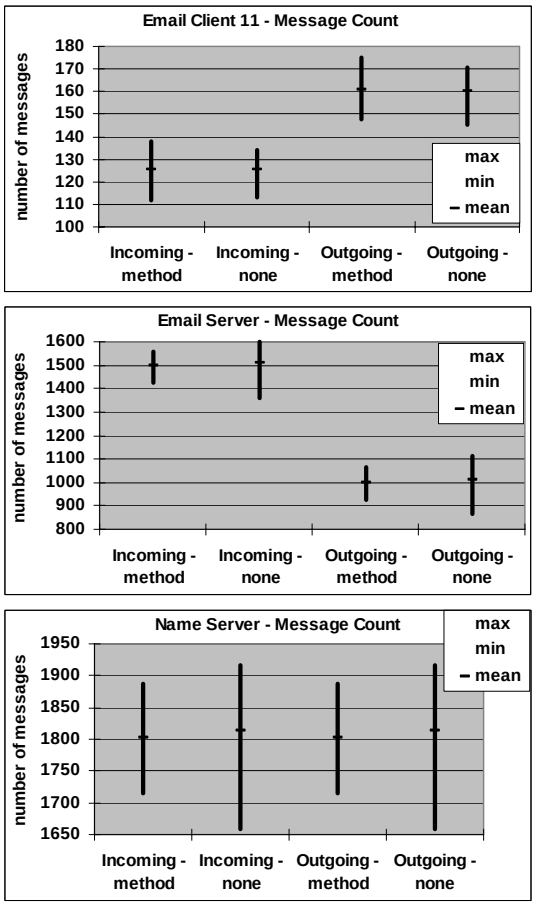


FIGURE 45. Email Application – Message Counts[9]

We expect the number of messages exchanged between DCO objects to vary between simulation runs. However, from a quantum perspective, we expect to see consistency on the mean number of messages as well as consistency on the range across the two simulation sets of runs for the two Name Server configurations. The plots in Figure 45 depict the message counts for incoming and outgoing traffic on three of the DCO software objects. These plots reflect this consistency.

[9] Note, the *none* and *method* labels in these plots reflect the thread mode setting of the Name Server object during the simulation runs.

The key configuration difference between the two sets of simulation runs is the thread mode on the Name Server object. When set to *none*, one and only one Name Server thread can process at any point in time, and new requests get queued for execution. When set to *method*, each incoming invocation or message results in a new Name Server thread starting immediately, and no requests get queued. The following plots depict various implications of this behavior on various components.

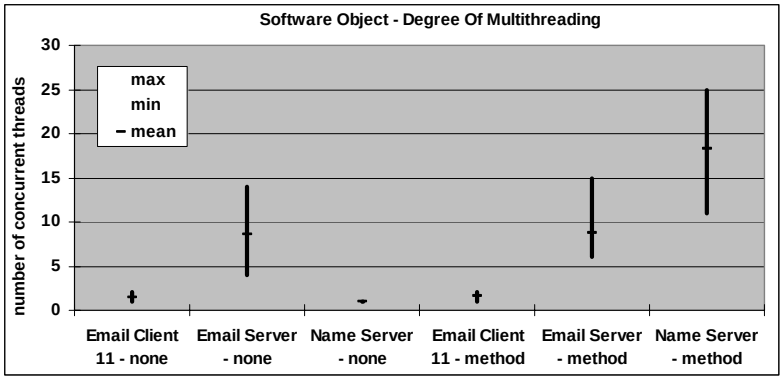


FIGURE 46. Email Application – Degree of Multithreading Maxima[10]

The *degree of multithreading* is a count of the number of concurrent jobs from a given software object. In Figure 46, one and only one thread is active when the Name Server is set to *none*. However, when it is set to *method*, it shows a maximum (for any one simulation run) multithreading range of 11 to 25 active threads. The multithreaded mode setting for the Name Server, however, shows no impact on the multithreading behavior of the Email Client or Email Server, as their respective multithreading behaviors are directly controlled by their own multithreaded mode settings, *object* and *method* level respectively.

[10] Note, the *none* and *method* labels in these plots reflect the thread mode setting of the Name Server object during the simulation runs.

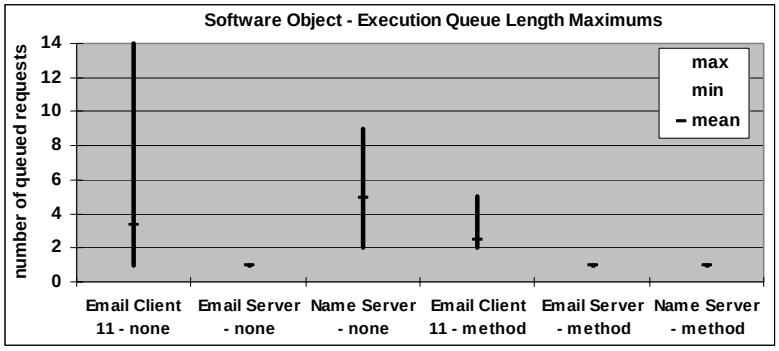


FIGURE 47. Email Application – Queue Length Maxima[11]

Figure 47 shows the impact of the Name Server thread mode setting on the maximum queue lengths seen in each software object during each simulation run. For the maximum queue length, the Name Server mode setting directly impacts the queuing behavior of itself. Yet, it also indirectly impacts the queuing behavior of the Email Client, which is set to an *object* level multithreaded mode. For the Name Server set to *none*, as the Email Client receives and processes incoming messages, Email Client jobs that are dependent on invocation requests to the Name Server get queued as those requests get queued at the Name Server. From this *none* scenario, we see the Email Client having maximum queue lengths in the range of 1 to 14. For the *method* level configuration, however, a queue at the Name Server does not grow – each incoming request being immediately turned around into a job –, and so there is a significantly lower opportunity for a queue to build at the Email Client. So, the Email Client maxima queue length drops to a range of 2 to 5.

[11] Note, the *none* and *method* labels in these plots reflect the thread mode setting of the Name Server object during the simulation runs.

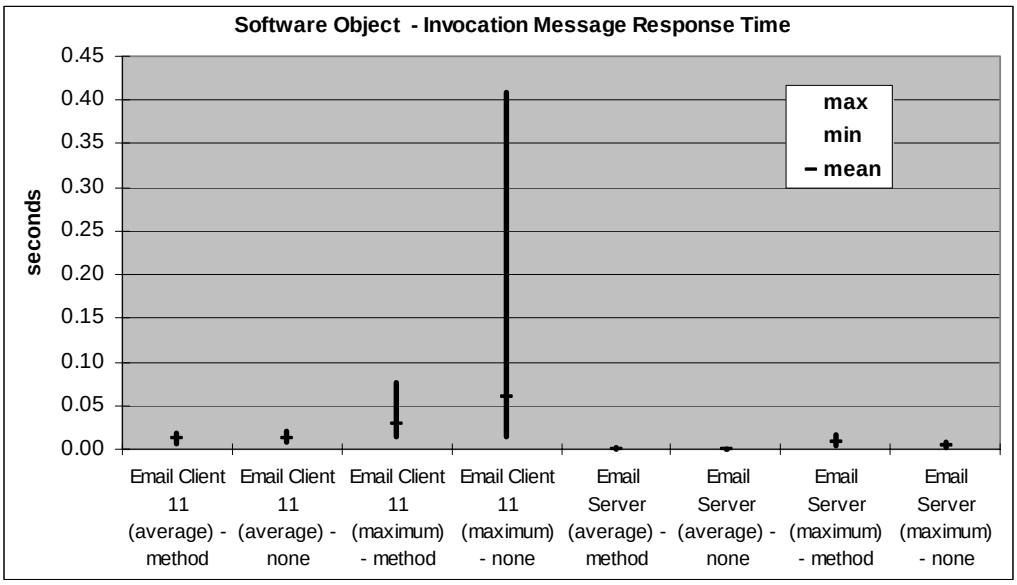


FIGURE 48. Email Application – Invocation Message Response Times[12]

In distributed systems, response time is often an issue of particular interest. For this case study, the response time of the Name Server to the Email Clients and Email Server is of interest. The results are plotted in Figure 48. These response time statistics reflect the time for an invocation request to transit the LCN, plus the time to process at the Name Server, plus the time for the response to return over the LCN.

During any given simulation, the average response time is computed over all the invocations a software object makes during that simulation. Similarly, the maximum response time is the one that takes the longest to get a response. Across the 20 simulations for a given Name Server configuration, Figure 48 plots the highest (max), lowest (min), and mean of the average response times collected during each simulation.

[12] Note, the *none* and *method* labels in these plots reflect the thread mode setting of the Name Server object during the simulation runs.

The figure also plots the highest (max), lowest (min), and mean of the maximum response times collected during individual simulation runs. For instance, the max, min, and mean of "Email Client 11 (average) - none" (plot column 2) are all approximately 0.01 seconds, whereas the max, min, and mean of the "Email Client 11 (maximum) - none" (plot column 4) are approximately 0.4, 0.06, and 0.02 seconds respectively. From these plots, we see that using the Name Server in the *method* thread mode has a negligible impact on the average response time, but that it can have a significant impact on reducing the maximum response time.

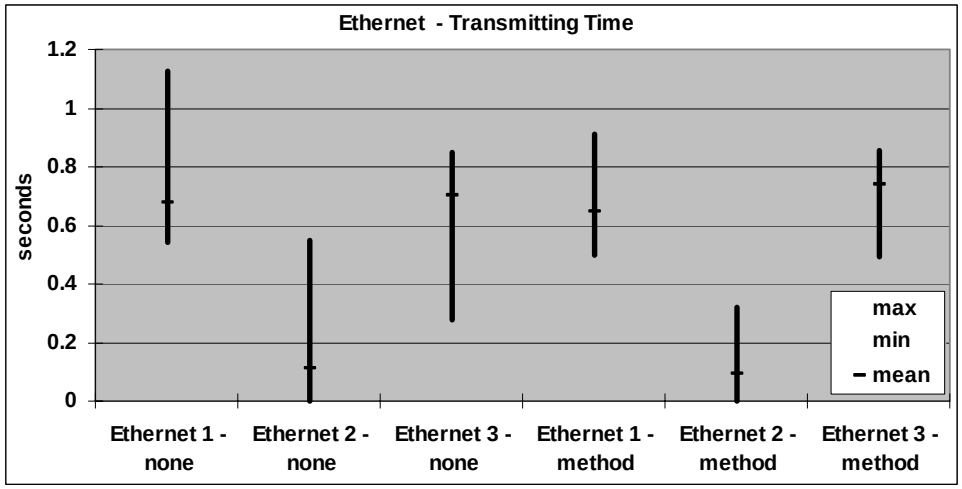


FIGURE 49. Email Application – Ethernet Transmission Times[13]

Figure 49 depicts transmission time statistics on the three ethernets providing connectivity between the routers. This transmission time is the amount of time the

[13] Note, the *none* and *method* labels in these plots reflect the thread mode setting of the Name Server object during the simulation runs.

ethernet was busy transmitting successful frames; in other words, it does not account for time the ethernet was idle or active with collisions.

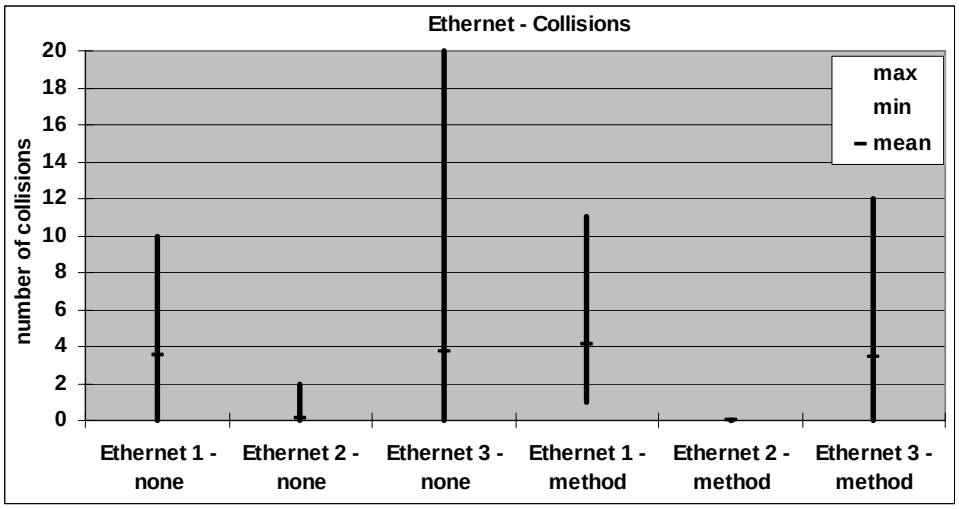


FIGURE 50. Email Application – Ethernet Collisions[14]

Extensive collisions on an ethernet can have significant impact on the throughput performance of a distributed system. Increasing the number of devices on an ethernet increases the probability of a collision, as does increasing the traffic load on the ethernet. In this case study, the acceptor (as part of the experimental frame) invokes all fifteen Email Clients simultaneously. Since all fifteen Email Clients and their associated processors exhibit the same behavior, collisions are highly likely to occur due to the simultaneous introduction of traffic on the LCN. Figure 50 plots the collision results on each of the three ethernet links, as collected during the two sets of simulation runs.

[14] Note, the *none* and *method* labels in these plots reflect the thread mode setting of the Name Server object during the simulation runs.

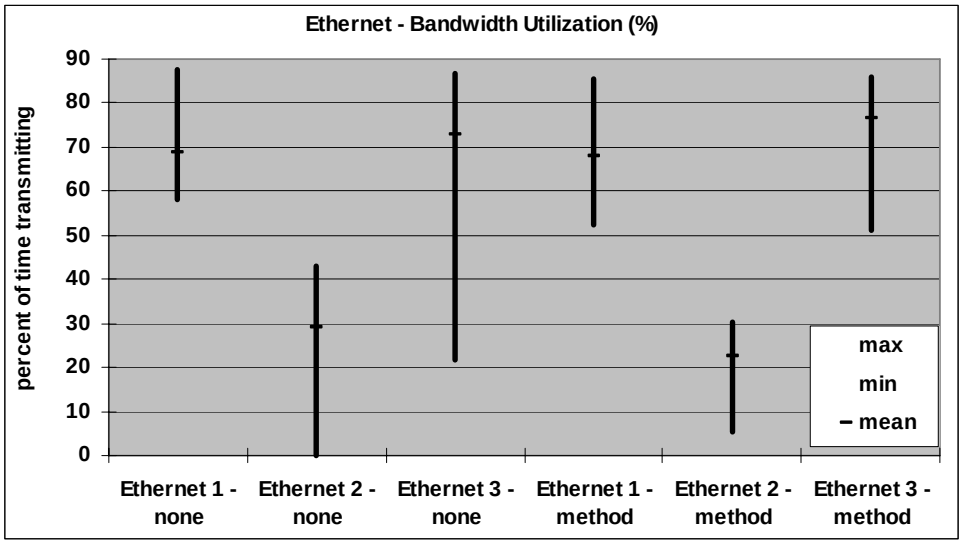


FIGURE 51. Email Application – Ethernet Bandwidth Utilization[15]

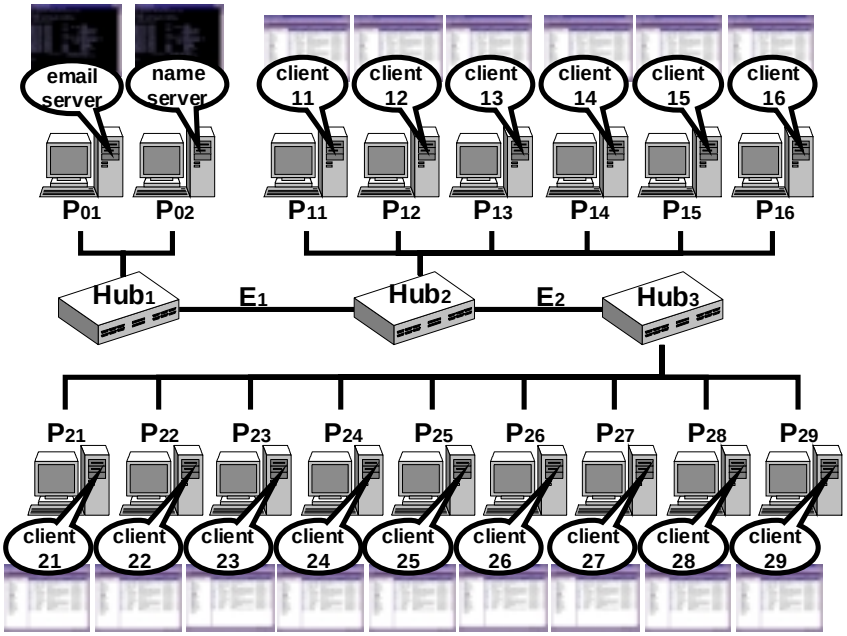
Bandwidth utilization is another metric that is often used in evaluating network performance and potential bottlenecks to system throughput. Figure 51 depicts the utilization measured during the email application simulations. Since this utilization metric is the percentage of time each ethernet spent successfully transmitting data, the results display the same qualitative trajectories we observed in Figure 49 for the ethernet transmission times.

6.4. Email Application LCN Alternatives

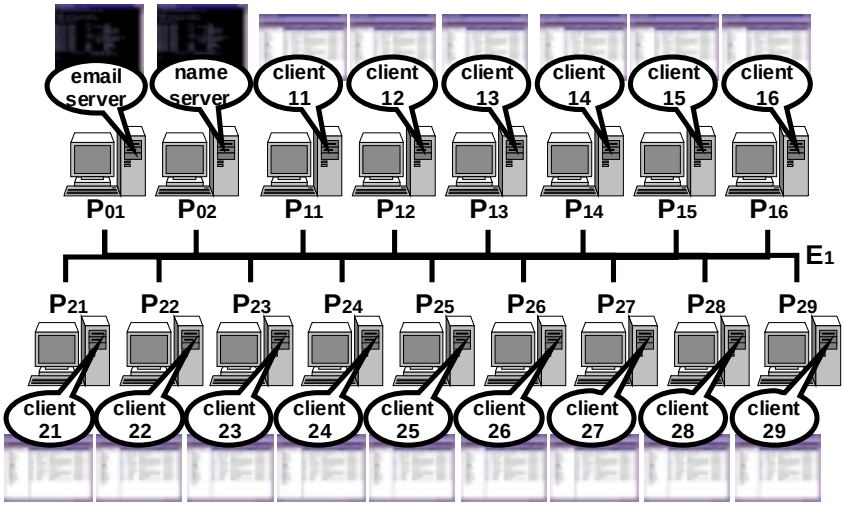
For this case study, we consider the same email application as described in the previous study, but now our focus is on alternative LCN configurations. In particular, the DCO and OSM models remain unchanged in this investigation; only the LCN and the

[15] Note, the *none* and *method* labels in these plots reflect the thread mode setting of the Name Server object during the simulation runs.

experimental frame (for LCN dependent components) are modified. This case study demonstrates the degree of independence between the LCN, DCO, and OSM models while signifying the interdependence of distributed systems behavior.



Alternative 1: Hub-Bus



Alternative 2: Bus

FIGURE 52. Email Application Alternative LCNs

We consider two LCN alternatives to the baseline Router-Star configuration of the original case study. For the first alternative, the Hub-Bus configuration, the LCN network consists of three ethernet hubs that provide "bus" connectivity to the three sets of processors. For the second alternative, the Bus configuration, the entire LCN network consists of a single shared ethernet "bus" link. The baseline Router-Star configuration is as depicted in Figure 42. The Hub-Bus and the Bus alternate LCN configurations are both depicted in Figure 52.

The experimental frame for each alternative is very similar across each alternative. Just as in the baseline configuration, an acceptor is coupled to each Email Client to invoke it forty times at the start of each of twenty simulation runs. Each simulation executes until all objects are passive. The software domain transducer monitors the entire collection of email application software objects. The Email Server, Name Server, and one of the Email Client software objects are each monitored with a software object transducer. Three processors are also monitored: P_{01}, P_{02}, and P_{11}. For the Router-Star configuration, the three ethernets interconnecting the routers are monitored. For the Hub-Star configuration, the two ethernets interconnecting the hubs are monitored. In the last alternative, the single ethernet link is monitored. Again, transducers collect system events from their components throughout each simulation run with statistical maxima, means, and minima collected and computed across the twenty simulation runs for each alternative and for each of the Name Server thread mode settings.

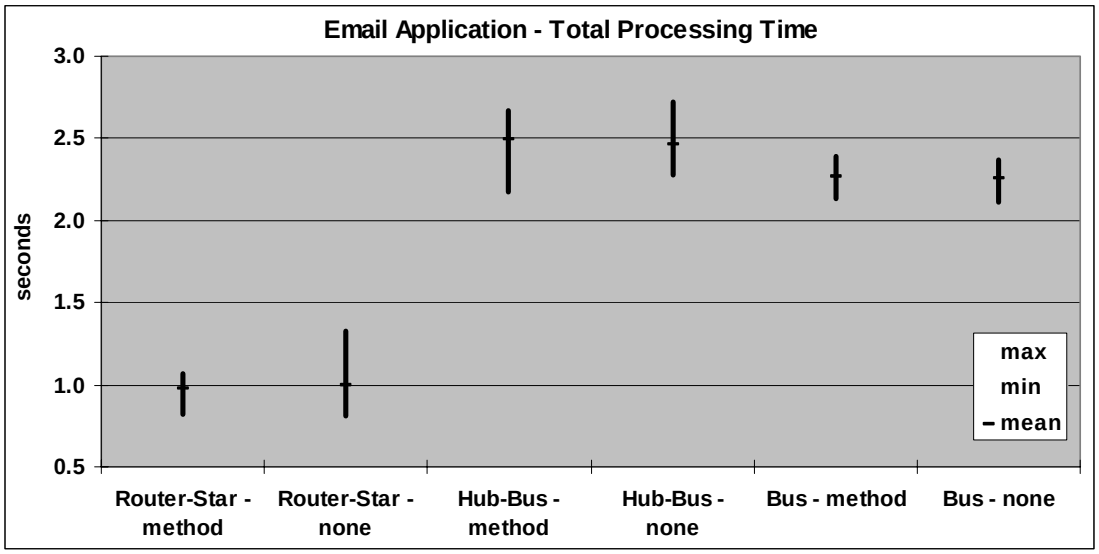


FIGURE 53. Email Application – Total Processing Time[16]

Figure 53 plots the total processing time maxima, means, and minima collected across the 20 simulation runs for each configuration. Within each alternative configuration, the thread mode setting for the Name Server has no significant impact on total processing times. The LCN configuration, however, has significant impact on the total processing times, with the baseline Router-Star configuration providing the shortest processing times. This is expected as the routers enable the traffic to be routed. In particular, the expectation is that ethernet E1 and E3 will load balance the LCN traffic between the Email Clients and the Email and Name Servers. We will confirm this as we examine the results further.

[16] Note, the *none* and *method* labels in these plots reflect the thread mode setting of the Name Server object during the simulation runs.

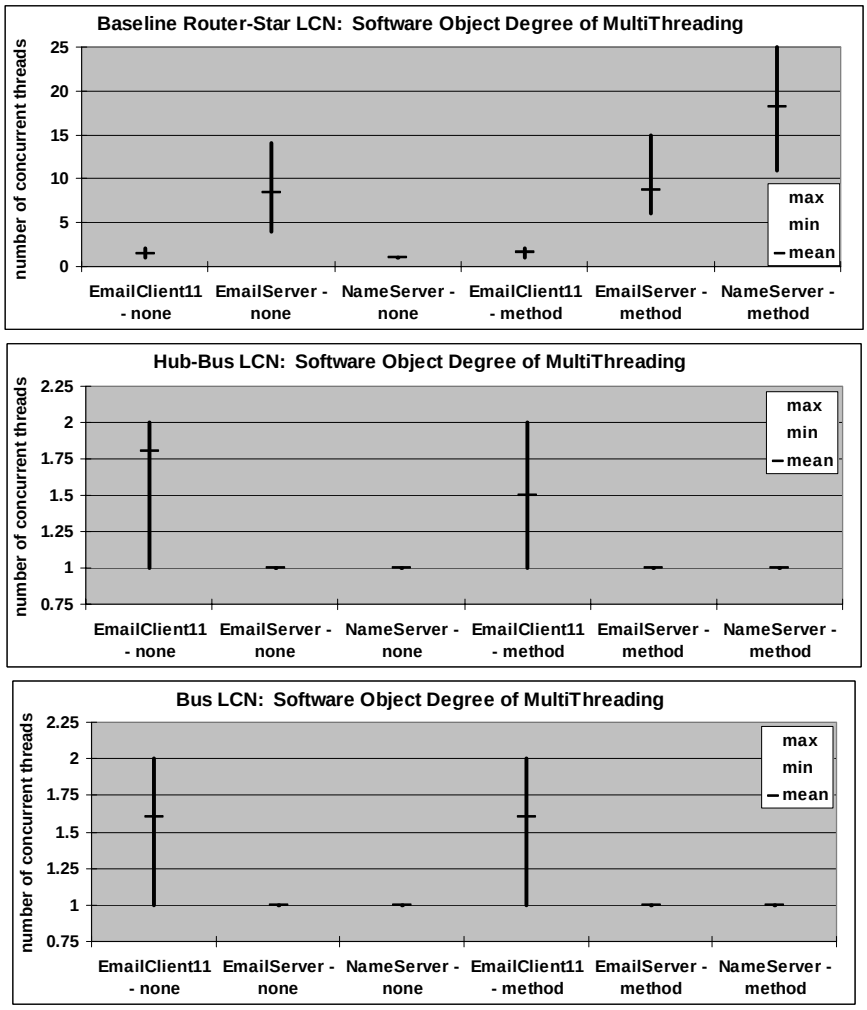


FIGURE 54. Email Application – Degree of Multithreading Maxima[17]

The *degree of multithreading* is a count of the number of concurrently active jobs from a given software object. The *degree of multithreading maxima* is a look at the maximum value this count achieved during a simulation run. For the three monitored software objects, the plots in Figure 54 show the range of maximum values observed over

[17] Note, the *none* and *method* labels in these plots reflect the thread mode setting of the Name Server object during the simulation runs.

the twenty simulation runs for each alternative configuration. The first plot in Figure 54 is of the baseline alternative and shows the same results depicted earlier in Figure 46.

The second and third plots reflect the results observed for LCN alternatives 1 and 2, respectively. While these two plots show no significant difference between each other, they are drastically different from the baseline Router-Star configuration. In the Hub-Bus and Bus configurations, the transit time for an interaction arc across the ethernet is longer than the computational time required for any method within the Email Server and Name Server software objects. Thus, the Email Server and Name Server only have a single thread active at any point in time during the simulation for these two configurations. In the baseline Router-Star configuration, however, the star configuration to the processors does not delay the interaction arcs as much, and the Email Server services several requests at once generating several concurrent threads. Concurrent threads in the Name Server, however, are dependent on the thread mode setting. When the Name Server is set to *none*, one and only one thread is active at any time for the Name Server. With the *method* setting in the Router-Star LCN configuration, the Name Server maximum multithreading range is 11 to 25 with a mean of 18.

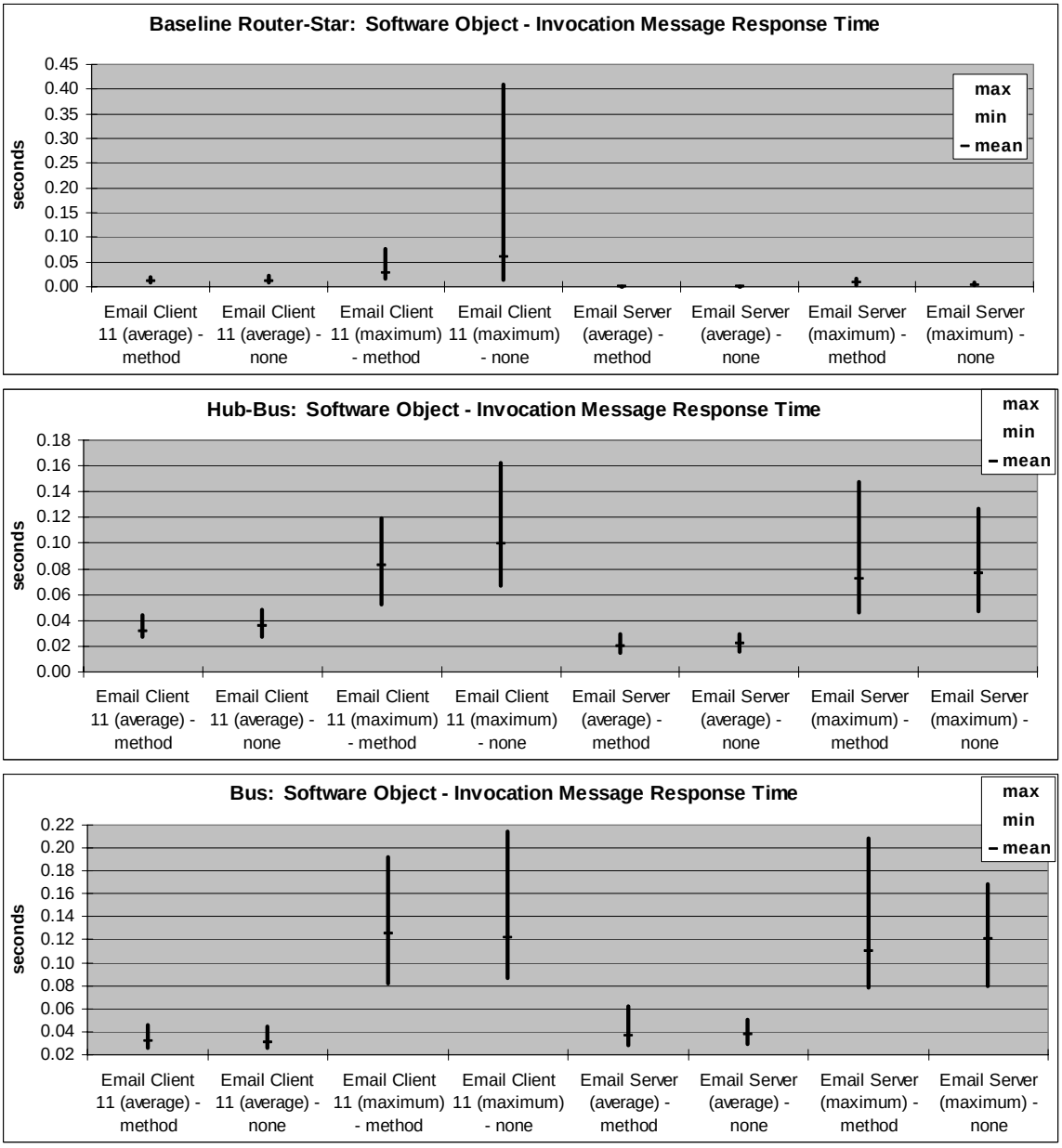


FIGURE 55. Email Application – Invocation Message Response Times[18]

[18] Note, the *none* and *method* labels in these plots reflect the thread mode setting of the Name Server object during the simulation runs.

The average and maximum *invocation message response times* for Email Client 11 and the Email Server are plotted in the graphs of Figure 55. For the baseline configuration, these are the same results as previously presented in Figure 48. In general, these results show that the Router-Star configuration offers a significantly better performance with respect to the average and worst case maximum response times.

While both alternative LCN configurations show the same basic behavior, closer inspection of the results reveals an interesting occurrence for the Email Client 11. While the average response times for Email Client 11 are lower in the Bus versus Hub-Bus configuration, the maximum response times tend to be higher in the Bus versus Hub-Bus configuration. This response time behavior reflects that the *average invocation request* typically experiences the transit delay of only one ethernet link in the Bus configuration and it typically experiences the transit delay of three ethernet links in the Hub-Bus configuration. In the worst case *maximum response time* scenario, the traffic loading is such that an invocation request will often experience contentions and collisions on the single ethernet, which often exceed the collective transit delay of the multiple ethernet links in the Hub-Star configuration.

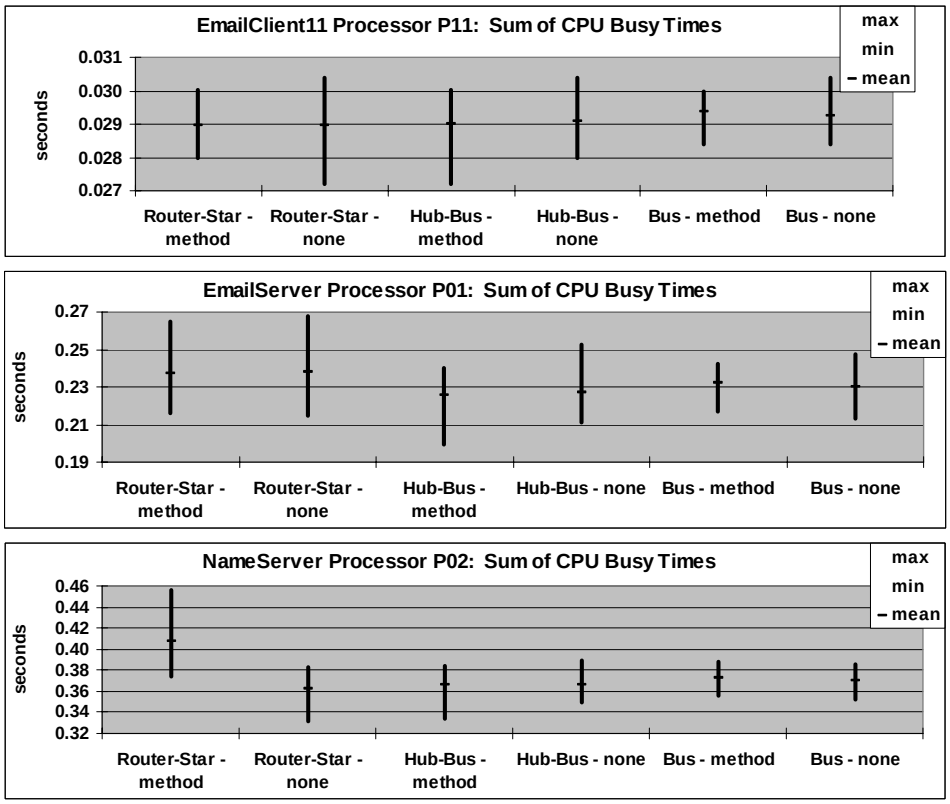


FIGURE 56. Email Application – Processor CPU Busy Times[19]

From our three processor transducers, we generate the plots depicted in Figure 56, which reflect the amount of time the processors were busy processing jobs during each simulation run. Noting the relative magnitude associated with each plot, the Name Server processor had the longest busy times, and the Email Client 11 processor had the shortest busy times. Since all the processors have equal cpu speed, this difference is attributable to the computational demands placed on each processor by the software objects. Since all fifteen Email Clients are configured to only interact with the Email and

[19] Note, the *none* and *method* labels in these plots reflect the thread mode setting of the Name Server object during the simulation runs.

Name Servers and the method work loads of all software objects are relatively equal, the Email and Name Server processors receive the higher computational demands causing these relatively longer cpu busy times. The Name Server processor is the "busiest" due to the configuration of the Email Clients to invoke and interact with the Name Server more often than with the Email Server.

In the first plot, no significant deviation in the Email Client processor P_{11} busy time is observed across the various configurations. In the second plot, the Email Server processor P_{01} shows higher cpu busy times in the Router-Star LCN configuration. In the third plot, the Name Server processor P_{02} shows a higher cpu busy time for the Router-Star configuration when the thread mode setting is *method*. These higher cpu busy times in these configurations are attributable to the multi-threading concurrency effects. In the LCN cpu model, a small overhead processing time tax is associated with each concurrent thread in the multi-tasking processor. As seen in Figure 54, a significantly larger number of concurrent threads occur for the Email Server and Name Server software objects in the Router-Star LCN configuration. This multi-threading concurrency effect is also seen in the concurrency of tasks that are running on these processors. The processor task concurrency results are depicted in Figure 57.

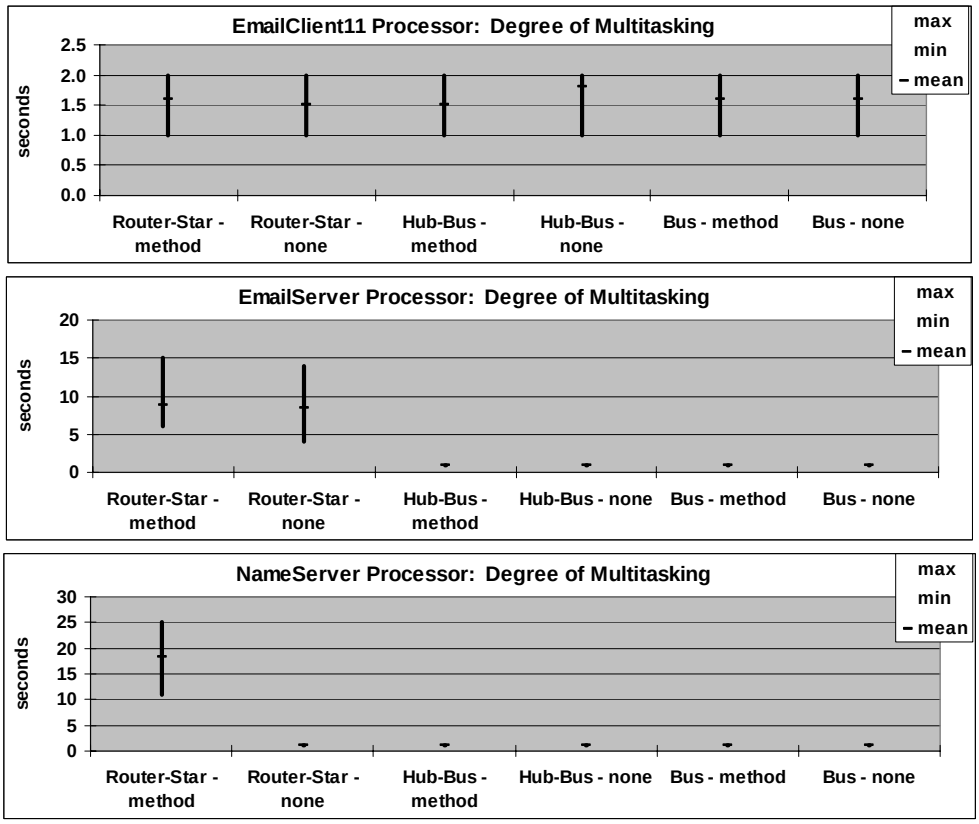


FIGURE 57. Email Application – Processor Degree of Multitasking[20]

The *degree of software multi-threading* depicts a measure of system concurrency from the software perspective. From the hardware perspective, this concurrency can be shown as the *degree of multi-tasking* in the LCN processors. Figure 57 shows the multi-tasking behaviors for the three observed processors. As only one software object is assigned to each processor in this case study, Figure 57 shows the same concurrency results as Figure 54.

[20] Note, the *none* and *method* labels in these plots reflect the thread mode setting of the Name Server object during the simulation runs.

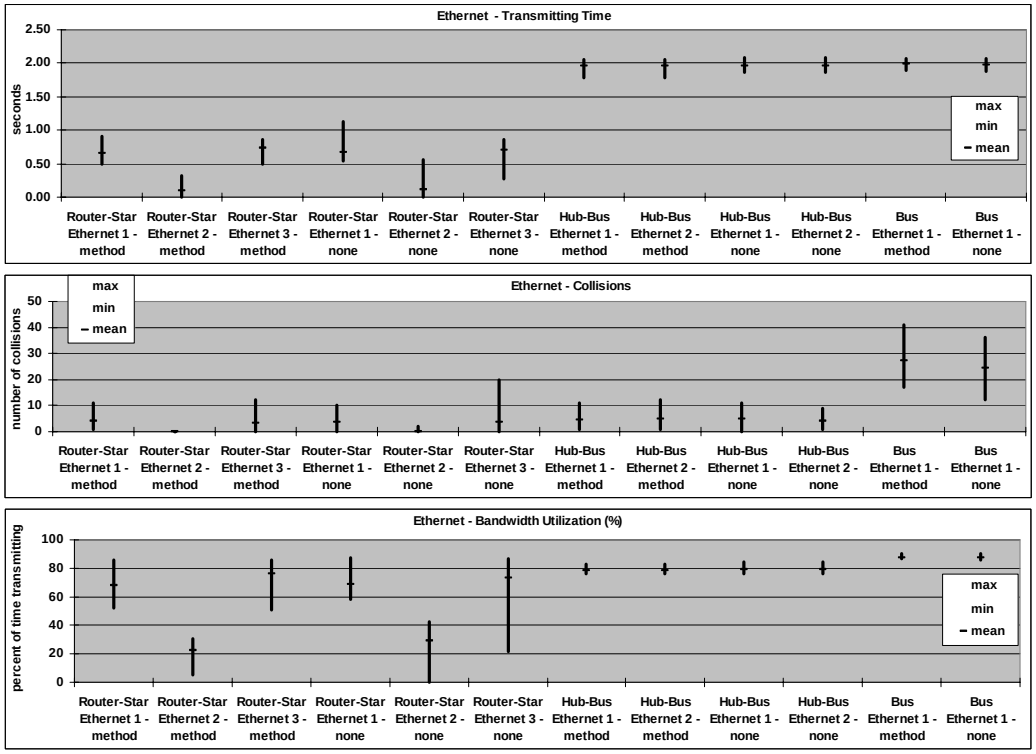


FIGURE 58. Email Application – Ethernet Link Performance[21]

The ethernet performance results for the main LCN links and each configuration are depicted in Figure 58. The *transmitting time* is the time the ethernet was busy with successful transmissions; it excludes ethernet collision time. The *collisions* graph reflects the maximum, mean, and minimum number of observed collisions. The *bandwidth utilization* is the percent of *transmitting time* over the *observation time*. The *observation time* starts with the first transmission over the link and ends with the completion of the last. As expected from the simulation execution time results in Figure 53, ethernet links E1 and E3 in the Router-Star configuration experience roughly equal traffic loads.

[21] Note, the *none* and *method* labels in these plots reflect the thread mode setting of the Name Server object during the simulation runs.

6.5. Case Studies Synopsis

This chapter details the results of four DEVS-DOC case studies. These case studies exhibit the power of the DEVS-DOC environment in modeling and simulating the dynamics of distributed computing systems. The first two studies use a *directed* modeling approach — specific software method and arc interaction sequences — and provide a comparison of DEVS-DOC simulations to real world results. The third and fourth studies use a *quantum* modeling approach providing a less rigid specification of software object behaviors.

In the first case study of the network management application, the key dynamics investigated involved the workload placed on the management application and the degree of concurrency exploited in processing that workload. The degree of concurrency was determined by the thread mode setting of the *manager* software object. The simulation results were fairly good for the *none* and *object* thread modes, but rather poor at the *method*-level. Analysis of these results identified a need for a better representation of the processor memory dynamics.

In the second case study, a DEVS/HLA-compliant distributed simulation federation is explored. In the real world system, the distributed performance effects of different predictive contract mechanisms are explored. In this DEVS-DOC case study, we investigate how to model the complex interactions of the DEVS/HLA software components and how well the simulation results reflect reality. From this experience, we discovered the utility of exploiting UML interaction diagrams to facilitate defining the DEVS-DOC software object interactions, which represent the DEVS/HLA software

component interactions in the real system. The implication here is the dual use of UML interaction diagrams to engineer the software system design and to simulate that design. The simulation results also show that, with only crude guesses (qualitative estimates) of computational loads associated with the software objects, generally correct performance behaviors emerge.

The third case study examined an email application scenario. This scenario had only three types of software objects — Email Clients, an Email Server, and a Name Server — and examined the impact of switching the thread mode of the Name Server. The simulation results of this study highlight the influence that this one setting can have on system performance as it impacts a system's overall processing time, the queuing of requests and jobs, and the response times experienced by clients of the service.

The fourth case study further analyzed the email application under alternative LCN configurations. Three alternative LCN configurations were investigated: Router-Star, Hub-Bus, and Bus. This study highlights the independence that exists in the LCN, DCO, and OSM modeling structures. In particular, only the LCN and its associated experimental frame required modification in exploring these alternatives. As expected, the simulation results confirmed that the Router-Star LCN configuration provides the best overall system performance.

7. CONCLUSION AND FUTURE WORK

In this dissertation, we have shown that:

> *A distributed systems modeling and simulation environment, based on a formal system specification methodology for developing abstract models of software behaviors and workloads, and abstract models of networked processing technologies and structures, can provide a practical means to describe and analyze Distributed Object Computing systems.*

This is demonstrated in the context of the DEVS-DOC modeling and simulation environment. DEVS-DOC models object-oriented software behaviors and networked computing technologies independently and then enables unifying them into a dynamic system of systems model. In chapter 3, we introduced the formal concepts behind the DOC ontology for the LCN, DCO, and OSM models; the DEVS formalism for specifying the DOC representations; and the Experimental Frame for prescribing simulation investigations. Chapter 4 detailed the structural implementation of the LCN, DCO, and OSM representations within the DEVS formalism. In chapter 5, we formally specified the behavioral dynamics developed for the LCN, DCO, and experimental frame components. In Chapter 6, we explored the representational power and practicality of the DEVS-DOC environment through four case studies. The first two case studies demonstrate the ability of DEVS-DOC to model real world distributed systems. The

second two studies demonstrate the structural independence and behavioral interdependence of the DEVS-DOC hardware and software abstractions. This chapter summarizes these contributions and outlines future work.

7.1. Contributions

The primary contribution of this thesis is the realization of a formal approach to modeling and simulating a distributed object computing system. This realization provides an example of how the dynamics of object-oriented software systems — the invocation and exchange of messages in concert with the computation of methods — can be modeled and joined with representations of networked computing technologies to construct a formal and practical system of systems specification.

Additionally, this thesis makes contributions to advances in modeling and simulation in general, and to the modeling and simulation of computing systems specifically. The general modeling and simulation contributions are the concept of a *layered experimental frame* to control and manage a series of simulations; the notion of *aggregated couplings* to enhance and simplify modeling portrayals of modular, hierarchical components; the introduction of the *distributed co-design* scheme; and a demonstration of the versatility of DEVS. Contributions to the modeling and simulation of distributed computing systems specifically fall into the areas of networked systems modeling, software modeling, and distributed systems modeling. These specific and general contributions are detailed below.

7.1.1. Networked Systems Modeling

We have shown how a formal, high-level representation of networking and computing technologies and components can support modeling distributed hardware architectures. Moreover, in the case studies we demonstrated how these components can be joined together to represent network topologies with the combined (components and topology) specification defining the loosely coupled network. The resulting LCN represents a distributed hardware architecture that imposes time and space constraints on associated software components.

Within the LCN component models, we have demonstrated a means to automatically register software components and develop routing tables. This automated routing table discovery service not only unburdens the DOC system modeler of these concerns but, eliminates potential problems in keeping routing tables consistent with OSM distribution alternatives. Thus, the automated routing table discovery approach avoids creating a dependency relationship between the LCN topology and the OSM distribution specification.

Current limitations within the LCN components are attributed to modeling granularity of the selected components. Three key examples are: the dynamics of cpu memory in response to the demands of associated software objects is limited; support for multiple communication modes and protocols is limited to one mechanism; and representation of link error rates and communication error recovery mechanisms require development.

7.1.2. Software Systems Modeling

We have shown how a formal, high-level representation of software components, following an object-oriented paradigm, can support modeling distributed software architectures. This representation accounts for computational loads associated with methods and data loads associated with object interactions. The object interactions have been shown to model synchronous and asynchronous client - server interactions as well as peer-to-peer messaging exchanges. We have also demonstrated how the multi-threading granularity of a software object can affect its performance and behavior.

This research has developed a means of modeling software independent of hardware imposed time and space constraints[22]. It is through the OSM distribution of the software objects onto the hardware components that such constraints impact the computing performance of software methods and the interaction performance of software object exchanges.

Two limitations exist within the DCO software representation. The first limitation is that it lacks mechanisms to support modeling software hierarchies. The current DCO software model only supports identifying peer-to-peer interactions; no constructs exists for identifying hierarchical relationships between the DCO components. The second limitation is a restricted representation of the semantics associated with the methods and interactions defined for a software component. Being able to associate semantic properties with a software object's methods and interactions, as demonstrated in [All97], would add a significant software architectural analysis capability.

7.1.3. Distributed Systems Modeling

The object system mapping component of this research presents an approach to explore issues of software distribution across networked hardware. In particular, the OSM enables investigating many inherent complexities associated with distributed object computing, such as computational load balancing, network traffic load balancing, and communications latency. Through the OSM, the systems modeler can support evaluation of the distribution alternatives to identify key performance tradeoffs and system performance optimizations.

7.1.4. Layered Experimental Frame

The contemporary experimental frame is a modeling and simulation artifact that specifies the conditions under which to observe and test a system for a given simulation. In this research, we expand on this idea with the specification of an experimental frame to control and observe a series of simulations. Under this scheme, an experimental frame is defined to control and collect data on a single simulation run; this artifact forms the base layer EF. An additional experimental frame can be specified to control and collect the results of a series of simulations. Such additional EF layers may control the initial simulation parameter settings, such as the number of pursuer-evader pairs in the DEVS/HLA Federation case study. Alternatively, the additional EF layers may collect results from a series of simulation runs to calculate performance and behavior statistics,

[22] The software object duty-cycle parameter is the only exception.

such as in the average, and maximum, *invocation message response time* in the Email Application case studies.

7.1.5. Aggregated Couplings

DEVS models have input and output ports, which enable coupling individual component models together to form system hierarchies. The fundamental DEVS coupling construct is an "add_coupling()" method, which is used to define couplings between one component and its output port and another component and its input port. Within the DEVS-DOC environment, each mapping of a DCO software object onto an LCN processor requires five of these fundamental DEVS coupling statements to fully specify the mapping. Similarly, the coupling of two LCN components into a network topological hierarchy may require multiple coupling statements, as may the coupling of DOC experimental frame components.

This research contributes the concept of *aggregated couplings* wherein the object-oriented *coupled* class model is extended with specific knowledge of the modeling domain. Such knowledge includes details on the types of components, the input and output ports associated with those components, and the allowed couplings of the output ports to input ports. With this domain specific coupling knowledge encoded, the systems modeler can maintain a focus at the higher level of abstraction of the component to component relations; the details of the individual port to port relations for such component relationships is handled automatically with the *aggregated coupling* knowledge.

7.1.6. Distributed Co-design

Contemporary co-design research has focused on tools and methods aiding the concurrent and cooperative design of the hardware and software elements of a target system. Often, these efforts focus on integrated approaches to specifications, designs, analyses, and simulations of the overall system under development. Systematic, computer aided approaches to requirements definition, specification, design, implementation, verification and validation are the objectives.

Through this research, we have introduced and developed a distributed aspect to co-design. This aspect provides a means to formally account for the distributed nature of networked hardware-software systems, which we call *Distributed Co-design*. Distributed Co-Design is defined as the activities to simultaneously, and collectively, design the hardware and software layers of a distributed system. The distributed hardware layer allows for exploring the design space for alternative high-level topologies and configurations, while the distributed software layer allows for exploration of the software design space from an object-oriented perspective. The independence maintained between the DEVS-DOC hardware and software representations allows the systems modeler to easily explore alternative distributions of the software objects across the hardware processors.

7.1.7. DEVS Versatility

DEVS is a system-theoretic based approach to the modeling of discrete-event systems. The DEVS modeling formalism enables characterizing systems in terms of

hierarchical modules with well-defined interfaces. The mathematical formality of DEVS focuses on the changes of state variables and generation of time segments that are piecewise constant. In essence, the formalism defines how to generate new state values and the times when these new values should take effect. An important aspect of the formalism is that the time intervals between event occurrences are variable.

Due to the system-theoretic foundations, the DEVS modeling paradigm is naturally rendered within object-oriented implementations. Consequently, DEVS has been implemented in sequential, parallel, and distributed environments. The DEVSJAVA implementation provides the benefits of the system-theoretic foundations, object-orientation, multi-threading concurrency, and simulation flexibility through the generation of traditional stand-alone applications or web-enabled applets.

The DEVS-DOC environment takes advantage of, and demonstrates the power of each of these features. The DEVS DCO and LCN component models demonstrate the suitability of the DEVS mathematical formalism to representing software, processing hardware, and communication systems, devices, and components. The DEVSJAVA object-orientation eases the structural construction and modularization of the LCN, DCO, OSM, and EF models.

The DEVSJAVA multi-threaded implementation permits simultaneous execution of several models, which aids the simulation of, and investigation of, concurrency issues among the DOC component models. Each model may be assigned its own thread, which is critical to the concurrent handling of the multiple internal and external events within the models.

Additionally, the DEVSJAVA implementation allows generation of traditional, stand-alone simulation applications or web-enabled applets. The generation of applets provides a means to run simulations within a browser. Web browser access provides a critical enabling mechanism for introducing a collaborative modeling and simulation environment. Enabling collaboration is a key feature to the realization of a Distributed Co-Design capability. Likewise, DEVSJAVA allows generation of stand-alone simulation applications. When simulation execution time becomes critical, these Java applications can then be processed through a Java optimizing compiler to improve execution performance on a target platform. Within DEVS-DOC, applets were used extensively for the development of the LCN and DCO component models, while stand-alone simulation applications were generated for the case study simulations.

7.2. Future Work

One basis for this work is the assumption that each processor will have its own unique memory. So, from a basic inquiry into the limitations of the proposed framework, support for system architectures of distributed shared-memory, object computing is inevitable and remains open for future research. A second basis of this work is the static distribution of software across hardware. Meanwhile, a growing part of today's computational needs are based on agent-oriented systems with inherently dynamic topological structures (e.g., mobile networked tele/video-conferences not only depend on varying network topologies, but also varying software computational characteristics such

as load and response time). Therefore, representation of variable-structure systems [Zei90] is believed to be of importance to future developments.

7.2.1. Real Time Distributed Object Computing

Increasingly, applications with stringent timing requirements are being implemented as distributed systems. These timing requirements mean that the underlying networks, operating systems, and middle-ware components forming the distributed system must be capable of providing quality of service (QoS) guarantees to the supported applications. Implementation mechanisms and technologies for such real time systems is an active area of research. A key issue in providing a QoS guarantee within a real-time system is resource utilization. DEVS-DOC provides a potential means to model and simulate proposed mechanisms and technologies and focus evaluation of resource utilization issues. To enable modeling of real time systems, DEVS-DOC will need to be extended with a means for DCO software objects to specify QoS requirements in methods and arcs and for LCN components to respond to QoS requests.

7.2.2. Mobile Software Agents

Mobile software agents are autonomous, intelligent programs that move through a network, searching for and interacting with services on a user's behalf. These systems use specialized servers to interpret the agent's behavior and communicate with other servers. A mobile agent has inherent navigational autonomy and can ask to be sent to other nodes. Mobile agents should be able to execute on every machine in a network, and the agent code should not have to be installed on every machine the agent could visit.

Therefore mobile agents use mobile code systems like Java with Java classes being loaded at runtime over the network. DEVS-DOC provides a potential means to model systems with mobile agents. Extensions would be needed in the DCO software objects to allow for an object to wrap and unwrap itself for firing across the LCN. The LCN also needs extensions to support dynamic routing of traffic to an agent that is moving or has moved.

7.2.3. Information Technologies and Standards

In [Car99], technologies are classified as either being sustaining or disruptive to successful business endeavors. In either case, information technologies and their application are continuously changing and evolving. To explore these technologies in the context of existing distributed systems or newly proposed systems, models of these technologies can be developed and coupled with existing DEVS-DOC components for investigation via simulation.

7.2.4. Integration

Integration of DEVS-DOC with other commercially supported modeling and simulation packages is another direction of interest. For example, several commercial packages support modeling and simulating communications networks, e.g., OPNET by MIL3 and COMNET by CACI. Rather than continue to develop and maintain LCN component models, the LCN components could be reconfigured to integrate into these existing commercially supported packages. The DCO and OSM abstractions would continue to be used to model the software architecture and map its objects onto the new

LCN components. These new LCN components would then translate arc firings into traffic loads for simulation through the commercial package. On the receiving side, the commercial package results would be translated by the LCN components back into arcs for delivery to target DCO software objects based on OSM mappings.

7.2.5. DEVS-DOC In Collaborative and Distributed Modeling and Simulation

At the University of Arizona, several projects are underway to extend *collaboration* support to modeling and simulation capabilities. These projects seek to build on experience with GroupSystems, a University of Arizona spin-off groupware environment [Nun95] in combination with advances in modeling and simulation methodology and high performance, distributed simulation support environments [Zei97b]. Research concepts evolving from these projects form the basis of a conceptual collaborative modeling and simulation architecture. This conceptual architecture is under investigation and the DEVS-DOC capability is being explored as a means to study alternative implementation designs for the collaboration architecture.

Figure 59 shows a component-based architecture for the DEVS Collaborative and Distributed Modeling and Simulation environment [Sar97, Sar99a, and Sar99b]. Components are broadly categorized into three layers. The lowest layer provides standard services such as persistent storage and middleware supporting distributed computing. In the middle layer, domain neutral modeling and simulation capabilities enable representations of hierarchical, heterogeneous models (of software and hardware). The Co-Design Modeling component provides *modeling constructs* enabling

representations of software objects, hardware components, and their associations (as detailed in this dissertation). Well-defined modeling constructs offered by the Co-Design Modeling component provide the basis for domain-dependent model development via model construction and composition components. The top layer role is to provide domain-dependent features built on the services provided by the lowest and middle layers. This architecture, based on the fundamental DEVS paradigm, facilitates extension of DEVS-DOC modeling to be used in collaborative settings with distributed simulation support.

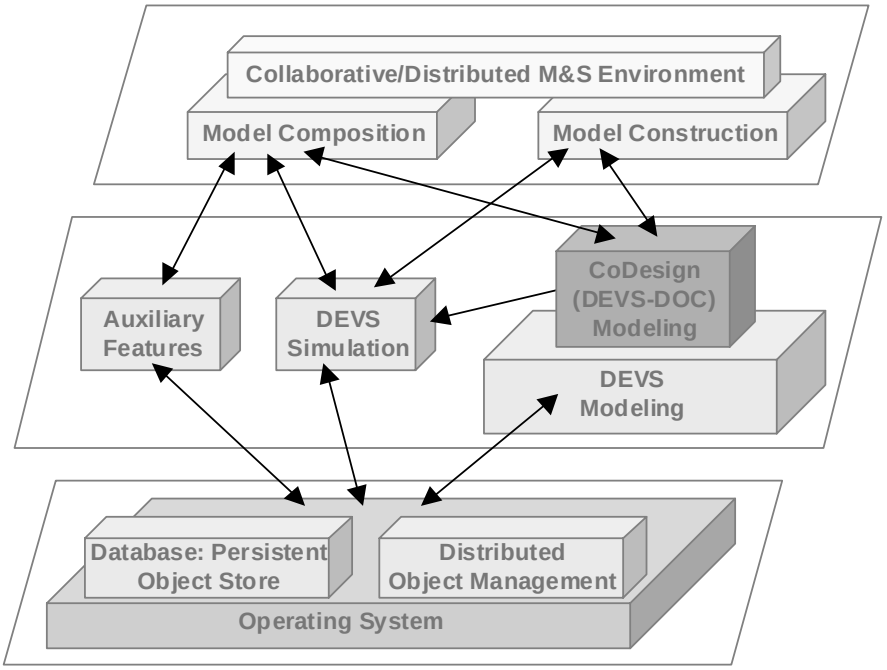


FIGURE 59. Collaborative/Distributed M&S Architecture

APPENDIX A. BUTLER'S DOC FORMALISM

A.1 Hardware: LCN Model

The formal LCN representation is a 5-tuple set H containing a set P of processors, a set K of network gates, a set N of network links, and mapping functions f and g of processors and gates, respectively, onto a power set of network links. This 5-tuple LCN set is summarized in the following equations.

$$H \equiv (P, K, N, f, g)$$

$$f : \pi \rightarrow L; \quad \pi \in P, \quad L \subseteq N, \quad L \neq \varnothing$$

$$g : \phi \rightarrow L; \quad \phi \in K, \quad L \subseteq N, \quad L \neq \varnothing$$

A processor π is an LCN component capable of performing computational work under the resource constraints of its processor speed S and its storage capacity D. The processor speed S is defined as a random variable since, during the processing of any given job, CPU time is that portion when it is not engaged in servicing overhead tasks, operating system daemons, and other processing requests external to the scope of the modeled simulation space. Similarly, the storage capacity D is a random variable accounting for swap space, operating system overhead, and other storage requirements outside the scope of the modeled simulation space. The following equation summarizes this representation.

$$\pi \equiv (S, D); \quad \pi \in P, \quad S > 0, \quad D > 0$$

A network gate ϕ is an LCN component capable of passing data traffic from an incident link to other incident links. Gates have two operational modes: *hub* or *router*. Hub mode results in all incident links passing the same data traffic. Router mode

switches inbound data trafffic from one incident link to one outbound link that provides direct or indirect connectivity to the targeted destination. Data traffic is processed through the gate based on constraints of buffer size R and bandwidth B. As R and B represent available resources to the system being modeled, they are defined as random variables due to the dynamics of external (to the model) conditions. Thus, a gate is defined as follows.

$$\phi \equiv (m, R, B); \quad \phi \in K, \quad m \in \{hub, router\}, \quad R \geq 0, \quad B > 0$$

A network link η is an LCN component connecting one or more processors and network gates to provide a communications path. A link may have one or more independent channels that have a bandwidth B. A link is also assigned an error coefficient ε (of units errors/second), which influences data retransmissions. The bandwidth B is a random variable in representing resources available to the system being modeled and excluding bandwidth consumed by external conditions. The error coefficient ε is a random variable to account for the non-deterministic nature of all real communications channels. A network link is defined with the following two equations.

$$\eta \equiv (\varepsilon, C); \quad \eta \in N, \quad \varepsilon \geq 0, \quad C \neq \varnothing$$

$$(i, B) \in C; \quad B > 0$$

A.2 Software: DCO Model

The formal DCO representation is a 9-tuple set S containing a set D of domains, a set T of software objects, a set A of directed invocation arcs, a set G of directed message arcs, and mapping functions u, v, x, y, and z. Function z maps software objects of T onto a power set of domains D. Functions x and y map the calling ends of invocation

arcs A onto software objects of T and the target ends of A onto a power set of T. Functions u and v map the source ends of message arcs G onto software objects of T and the target ends of A onto a power set of T. This representation is summarized in the following six equations.

$$S \equiv (D, T, A, G, u, v, x, y, z)$$

$$u : \gamma \rightarrow \tau; \quad \gamma \in G, \quad \tau \in T$$

$$v : \gamma \rightarrow L; \quad \gamma \in G, \quad L \subseteq T, \quad |L| \geq 2$$

$$x : \alpha \rightarrow \tau; \quad \alpha \in A, \quad \tau \in T$$

$$y : \alpha \rightarrow L; \quad \alpha \in A, \quad L \subseteq T, \quad L \neq \varnothing$$

$$z : \tau \rightarrow L; \quad \tau \in T, \quad L \subseteq D, \quad L \neq \varnothing$$

A domain δ is a DCO component representing a set of software objects that form an independently executable program. Software objects may be mapped to more than one domain. Domains serve two purposes, organizing software objects for the extraction of simulation data and for scheduling program initiations. For scheduling program initiations, selected objects in a domain are defined as *initializer* objects Q and assigned a duty cycle U representing the time between program executions. The duty cycle U is set as a random variable.

$$\delta \equiv (Q, U); \quad \delta \in D, \quad Q \subseteq \{ \tau; z(\tau)=\delta \}, \quad U > 0$$

A software object τ is a DCO component representing a software object in the traditional object-oriented concept of an object, composed of *attributes* defining the objects state and a set of *methods* that operate on the attributes. Software objects are formally defined with a 4-tuple set τ. The object's size C refers to its collective memory allocation requirement. The object has a thread mode m specifying the granularity of

multithreading behavior to be object, method, or none (single thread capable only). Each method of the software object is characterized by a computational work load factor x and an invocation probability ρ. In a quantum simulation, it becomes irrelevant which method is invoked on a given invocation; rather, methods need to be invoked in correct proportion across the aggregation of invocations on the object.

$$\tau \equiv (C, m, X, P); \quad \tau \in T, \quad C > 0, \quad m \in \{ \text{Object, Method, None} \}$$

$$X \equiv < x_1, \ldots, x_n >; \quad x_i > 0, \quad 1 \le i \le n$$

$$P \equiv < \rho_1, \ldots, \rho_n >; \quad \rho_i \ge 0, \quad \Sigma\rho_i = 1, \quad 1 \le i \le n$$

An invocation arc α is a DCO component representing client-server request and response exchanges between calling (client) and called (server) software objects. Invocation arcs have a firing frequency F, a request size P, a response size R, and a blocking mode b. The firing frequency F is a random variable that is dependent on the computational progress of the source software object, i.e., how much work must be completed between invocations. The invocation request size P and response size R are both random variables. The blocking mode b may be either *Synchronous* or *Asynchronous*. For synchronous invocations, the client object is blocked until the server response is received, while asynchronous invocations allow the client object to continue processing.

$$\alpha \equiv (F, P, R, b); \quad \alpha \in A, \quad F > 0, \quad P > 0, \quad R \ge 0, \quad b \in \{ \text{Sync, Async} \}$$

A message arc γ is a DCO component representing peer-to-peer message passing between a source software object and a set of destination objects. The message arc is defined as having a firing frequency F and a message size M. The firing frequency F is a

random variable that is dependent on the computational progress of the source software object, i.e., how much work must be completed between messages. The message size M is a random variable denoting the size of the message being exchanged.

$$\gamma \equiv (F, M); \quad \gamma \in G, \quad F > 0, \quad M > 0$$

A.3 Distribution: OSM Model

The formal OSM representation is a 5-tuple set Ψ containing an LCN representation H, a DCO representation S, a set of communication modes C, and mapping functions λ and μ. Function λ maps DCO software objects T_S onto LCN processors P_H. Function μ maps DCO invocation arcs A_S and message arcs G_S onto communication modes C. A communication mode c is defined by random variables representing packet size P, packet overhead V, and acknowledgment packet size R. This representation is summarized in the following four equations.

$$\Psi \equiv (H, S, \lambda, C, \mu)$$

$$\lambda : \tau \rightarrow \pi; \quad \forall \tau \in T_S, \quad \pi \in P_H$$

$$c = (P, V, R); \quad c \in C, \quad P > V \geq 0, \quad R \geq 0$$

$$\mu : \omega \rightarrow c; \quad \forall \omega \in \{ A_S \cup G_S \}, \quad c \in C$$

APPENDIX B. CASE STUDY CODE

B.1 Simple Network Management Protocol (SNMP) Monitoring

```
/*
 *  Filename   : snmp.java
 *  Version    : 1.0
 *  Date       : 01-13-99
 *  Author     : Daryl Hild
*/

//package DOCapp;

import DOC.*;
import Zdevs.*;
import Zdevs.Zcontainer.*;

public class snmp extends digraphDCO {

   //simulation run parameters
   protected String threadMode;
   protected int    loopCount, simRuns;
   protected double simTime, Mem, BFhub, BFmau;

   //system under study components
   protected atomic  loop, mgr,
      agent4,   mau4,    link4,   agent6,   mau6,    link6,
      agent15,  mau15,   link15,  agent17,  mau17,   link17,
      agent19,  mau19,   link19;
   protected digraph asc4,     asc6,    asc15,    asc17,    asc19;
   protected atomic  hub;

   //experimental frame components
   protected atomic  tLoop,     tMgr,
                     tAgent4,   tAsc4,   tAgent6,   tAsc6,   tLink6,
                     tAsc_snmp, tTuple,  acceptor;
   protected set  snmpSW, processors;

public snmp() {
   super("snmp");
   threadMode="object"; loopCount=20;
   simTime=500; Mem=1E+4; BFhub=2E+6; BFmau=2E+6; simRuns=10;

   Loosely_Coupled_Network();
   Distributed_Cooperative_Objects();
   Object_System_Mapping();
   Experimental_Frame();

   initialize();
}
```

```
public void Loosely_Coupled_Network() {
   //declare LCN components
   double udpHS    = 8*8;      //udp header size in bytes * 8 bits/byte
   double ipHS     = 20*8;     // ip header size in bytes * 8 bits/byte
   double LS       = 2E+9;     //proc nic speed bits/sec
   double BW       = INFINITY; //proc nic bandwidth bits/sec
   double BFinf    = INFINITY; //buffer size in bytes * 8 bits/byte
   double BF_hub   = BFhub*8;  //buffer size in bytes * 8 bits/byte
   double BF_mau   = BFmau*8;  //buffer size in bytes * 8 bits/byte
   double cpuSp    = 200E+6;   //cpu speed ops/sec
   double memSz    = Mem*8;    //cpu mem in bytes * 8 bits/byte
   double swapTime =  1.1;     //cpu-mem swapTimePenalty
   double ES       = 10E+6;    //ethernet speed bits/sec
   int    etherSeg = 5;        //number of Ethernet Segments
   int    MPS      = 1500;     //maxPktSize

   //processors
   asc4=new processor("asc4",udpHS+ipHS,LS,BW,BFinf,cpuSp,memSz,MPS);
   add(asc4);
   asc6=new processor("asc6",udpHS+ipHS,LS,BW,BFinf,cpuSp,memSz,
      swapTime,MPS);
   add(asc6);
   asc15=new processor("asc15",udpHS+ipHS,LS,BW,BFinf,cpuSp,memSz,MPS);
   add(asc15);
   asc17=new processor("asc17",udpHS+ipHS,LS,BW,BFinf,cpuSp,memSz,MPS);
   add(asc17);
   asc19=new processor("asc19",udpHS+ipHS,LS,BW,BFinf,cpuSp,memSz,MPS);
   add(asc19);
   //processor mau's
   mau4  = new hub_ethernet("mau4",  ES, BW, BF_mau);   add(mau4);
   mau6  = new hub_ethernet("mau6",  ES, BW, BF_mau);   add(mau6);
   mau15 = new hub_ethernet("mau15", ES, BW, BF_mau);   add(mau15);
   mau17 = new hub_ethernet("mau17", ES, BW, BF_mau);   add(mau17);
   mau19 = new hub_ethernet("mau19", ES, BW, BF_mau);   add(mau19);
   //ethernet links
   link4  = new link_ethernet("link4",  etherSeg);   add(link4);
   link6  = new link_ethernet("link6",  etherSeg);   add(link6);
   link15 = new link_ethernet("link15", etherSeg);   add(link15);
   link17 = new link_ethernet("link17", etherSeg);   add(link17);
   link19 = new link_ethernet("link19", etherSeg);   add(link19);
   //hub
   queue etherSpeeds = new queue();
   for(int i=1; i<6; i++) etherSpeeds.add(new doubleEnt(ES));
   hub = new hub_ethernet("hub", etherSpeeds, BW, BF_hub);   add(hub);

   // couple LCN
   Add_coupling_LCNtoEthernet( asc4,   mau4,   link4);
   Add_coupling_LCNtoEthernet( asc6,   mau6,   link6);
   Add_coupling_LCNtoEthernet( asc15,  mau15,  link15);
   Add_coupling_LCNtoEthernet( asc17,  mau17,  link17);
   Add_coupling_LCNtoEthernet( asc19,  mau19,  link19);

   Add_coupling_LCNtoEthernet( hub,1,  link4);
```

```
   Add_coupling_LCNtoEthernet( hub,2,  link6);
   Add_coupling_LCNtoEthernet( hub,3,  link15);
   Add_coupling_LCNtoEthernet( hub,4,  link17);
   Add_coupling_LCNtoEthernet( hub,5,  link19);
}


public void Distributed_Cooperative_Objects() {
   //declare DCO components
   dcoArc noArc = new dcoArc();

   set methods = new set();
   set iArcs   = new set();
   set mArcs   = new set();

   //loop
   int loop_WL  = 2103792;
   int no_WL    =      10;
   //loop
   // loop methods:  methodName, queue_of(pairs_of(workLoad, arc))
   queue lp = new queue();
   //msgingArc:arcName,dstAddr,msgSize,returnSize,mmsgType,methodCalled
   lp.add(new task(no_WL,  new dcoArc("snmpwalk asc4",
"mgr",48*8,"message","snmpwalk asc4")));
   lp.add(new task(no_WL,  new dcoArc("snmpwalk asc6",
"mgr",48*8,"message","snmpwalk asc6")));
   lp.add(new task(no_WL,  new dcoArc("snmpwalk
asc15","mgr",48*8,"message","snmpwalk asc15")));
   lp.add(new task(no_WL,  new dcoArc("snmpwalk
asc17","mgr",48*8,"message","snmpwalk asc17")));
   lp.add(new task(no_WL,  new dcoArc("snmpwalk
asc19","mgr",48*8,"message","snmpwalk asc19")));
   // messagingArc:  arcName, dstAddr, msgSize, methodCalled
   methods = new set();
   methods.add(new method("loop", lp));
   //swObject: name,size,threadMode,methods,arcs,dutyCycle,initMethod
   loop = new swObject("loop",32000*8,"method",methods,new
set(),INFINITY,"loop");
   add(loop);

   //mgr
   int mgrSize = 32000*8; //bytes * 8 bits/byte
   int    snmpget_WL = 13160000;
   double timeOut    = 0.8;
   // mgr methods:  methodName, queue_of(pairs_of(workLoad, arc))
   queue snmpwalk4 = new queue();
   snmpwalk4.add(new task(snmpget_WL,new dcoArc("snmpget asc4
1","agent4",42*8,84*8,"invokeSync","snmpget",timeOut)));
   snmpwalk4.add(new task(snmpget_WL,new dcoArc("snmpget asc4
2","agent4",44*8,54*8,"invokeSync","snmpget",timeOut)));
   snmpwalk4.add(new task(snmpget_WL,new dcoArc("snmpget asc4
3","agent4",44*8,48*8,"invokeSync","snmpget",timeOut)));
```

```
   snmpwalk4.add(new task(snmpget_WL,new dcoArc("snmpget asc4
4","agent4",44*8,44*8,"invokeSync","snmpget",timeOut)));
   snmpwalk4.add(new task(snmpget_WL,new dcoArc("snmpget asc4
5","agent4",44*8,48*8,"invokeSync","snmpget",timeOut)));
   snmpwalk4.add(new task(snmpget_WL,new dcoArc("snmpget asc4
6","agent4",44*8,44*8,"invokeSync","snmpget",timeOut)));
   snmpwalk4.add(new task(snmpget_WL,new dcoArc("snmpget asc4
7","agent4",44*8,45*8,"invokeSync","snmpget",timeOut)));
   snmpwalk4.add(new task(snmpget_WL,new dcoArc("snmpget asc4
8","agent4",44*8,45*8,"invokeSync","snmpget",timeOut)));
   queue snmpwalk6 = new queue();
   snmpwalk6.add(new task(snmpget_WL,new dcoArc("snmpget asc6
1","agent6",39*8,81*8,"invokeSync","snmpget",timeOut)));
   snmpwalk6.add(new task(snmpget_WL,new dcoArc("snmpget asc6
2","agent6",41*8,51*8,"invokeSync","snmpget",timeOut)));
   snmpwalk6.add(new task(snmpget_WL,new dcoArc("snmpget asc6
3","agent6",41*8,45*8,"invokeSync","snmpget",timeOut)));
   snmpwalk6.add(new task(snmpget_WL,new dcoArc("snmpget asc6
4","agent6",41*8,47*8,"invokeSync","snmpget",timeOut)));
   snmpwalk6.add(new task(snmpget_WL,new dcoArc("snmpget asc6
5","agent6",41*8,45*8,"invokeSync","snmpget",timeOut)));
   snmpwalk6.add(new task(snmpget_WL,new dcoArc("snmpget asc6
6","agent6",41*8,41*8,"invokeSync","snmpget",timeOut)));
   snmpwalk6.add(new task(snmpget_WL,new dcoArc("snmpget asc6
7","agent6",41*8,42*8,"invokeSync","snmpget",timeOut)));
   snmpwalk6.add(new task(snmpget_WL,new dcoArc("snmpget asc6
8","agent6",41*8,42*8,"invokeSync","snmpget",timeOut)));
   snmpwalk6.add(new task(snmpget_WL,new dcoArc("snmpget asc6
9","agent6",41*8,42*8,"invokeSync","snmpget",timeOut)));
   queue snmpwalk15 = new queue();
   snmpwalk15.add(new task(snmpget_WL,new dcoArc("snmpget asc15
1","agent15",39*8,64*8,"invokeSync","snmpget",timeOut)));
   snmpwalk15.add(new task(snmpget_WL,new dcoArc("snmpget asc15
2","agent15",41*8,50*8,"invokeSync","snmpget",timeOut)));
   snmpwalk15.add(new task(snmpget_WL,new dcoArc("snmpget asc15
3","agent15",41*8,45*8,"invokeSync","snmpget",timeOut)));
   snmpwalk15.add(new task(snmpget_WL,new dcoArc("snmpget asc15
4","agent15",41*8,67*8,"invokeSync","snmpget",timeOut)));
   snmpwalk15.add(new task(snmpget_WL,new dcoArc("snmpget asc15
5","agent15",41*8,55*8,"invokeSync","snmpget",timeOut)));
   snmpwalk15.add(new task(snmpget_WL,new dcoArc("snmpget asc15
6","agent15",41*8,82*8,"invokeSync","snmpget",timeOut)));
   snmpwalk15.add(new task(snmpget_WL,new dcoArc("snmpget asc15
7","agent15",41*8,42*8,"invokeSync","snmpget",timeOut)));
   snmpwalk15.add(new task(snmpget_WL,new dcoArc("snmpget asc15
8","agent15",41*8,42*8,"invokeSync","snmpget",timeOut)));
   queue snmpwalk17 = new queue();
   snmpwalk17.add(new task(snmpget_WL,new dcoArc("snmpget asc17
1","agent17",39*8,100*8,"invokeSync","snmpget",timeOut)));
   snmpwalk17.add(new task(snmpget_WL,new dcoArc("snmpget asc17
2","agent17",41*8,60*8,"invokeSync","snmpget",timeOut)));
   snmpwalk17.add(new task(snmpget_WL,new dcoArc("snmpget asc17
3","agent17",41*8,45*8,"invokeSync","snmpget",timeOut)));
```

```
   snmpwalk17.add(new task(snmpget_WL,new dcoArc("snmpget asc17
4","agent17",41*8,41*8,"invokeSync","snmpget",timeOut)));
   snmpwalk17.add(new task(snmpget_WL,new dcoArc("snmpget asc17
5","agent17",41*8,46*8,"invokeSync","snmpget",timeOut)));
   snmpwalk17.add(new task(snmpget_WL,new dcoArc("snmpget asc17
6","agent17",41*8,41*8,"invokeSync","snmpget",timeOut)));
   snmpwalk17.add(new task(snmpget_WL,new dcoArc("snmpget asc17
7","agent17",41*8,42*8,"invokeSync","snmpget",timeOut)));
   snmpwalk17.add(new task(snmpget_WL,new dcoArc("snmpget asc17
8","agent17",41*8,42*8,"invokeSync","snmpget",timeOut)));
   queue snmpwalk19 = new queue();
   snmpwalk19.add(new task(snmpget_WL,new dcoArc("snmpget asc19
1","agent19",42*8,115*8,"invokeSync","snmpget",timeOut)));
   snmpwalk19.add(new task(snmpget_WL,new dcoArc("snmpget asc19
2","agent19",44*8,54*8,"invokeSync","snmpget",timeOut)));
   snmpwalk19.add(new task(snmpget_WL,new dcoArc("snmpget asc19
3","agent19",44*8,48*8,"invokeSync","snmpget",timeOut)));
   snmpwalk19.add(new task(snmpget_WL,new dcoArc("snmpget asc19
4","agent19",44*8,44*8,"invokeSync","snmpget",timeOut)));
   snmpwalk19.add(new task(snmpget_WL,new dcoArc("snmpget asc19
5","agent19",44*8,44*8,"invokeSync","snmpget",timeOut)));
   snmpwalk19.add(new task(snmpget_WL,new dcoArc("snmpget asc19
6","agent19",44*8,44*8,"invokeSync","snmpget",timeOut)));
   snmpwalk19.add(new task(snmpget_WL,new dcoArc("snmpget asc19
7","agent19",44*8,45*8,"invokeSync","snmpget",timeOut)));
   snmpwalk19.add(new task(snmpget_WL,new dcoArc("snmpget asc19
8","agent19",44*8,45*8,"invokeSync","snmpget",timeOut)));
   int methodMemLoad = 716800*8;//bytes * 8 bits/byte
   methods = new set();
   methods.add(new method("snmpwalk asc4", snmpwalk4, methodMemLoad));
   methods.add(new method("snmpwalk asc6", snmpwalk6, methodMemLoad));
   methods.add(new method("snmpwalk asc15",snmpwalk15,methodMemLoad));
   methods.add(new method("snmpwalk asc17",snmpwalk17,methodMemLoad));
   methods.add(new method("snmpwalk asc19",snmpwalk19,methodMemLoad));
   //swObject: name,size,threadMode,methods,arcs,dutyCycle,initMethod
   mgr = new swObject("mgr",mgrSize,threadMode,methods,new
set(),INFINITY,"snmpwalk");
   add(mgr);

   //Agents
   int agentSize            = 32000*8; //bytes * 8 bits/byte
   int snmpget_response_WL  = 2470600;
   int snmptrap_WL          = 1560000;
   // Agent methods:  methodName, queue_of(pairs_of(workLoad, arc))
   queue snmpget = new queue();
   snmpget.add(new task(snmpget_response_WL,noArc));
   queue snmptrap = new queue();
   snmptrap.add(new task(snmptrap_WL,  new
dcoArc("snmptrap","mgr",48*8,"message","snmptrap")));
   methods = new set();
   methods.add(new method("snmpget",  snmpget));
   methods.add(new method("snmptrap", snmptrap));
   //swObject: name,size,threadMode,methods,arcs,dutyCycle,initMethod
```

```
   agent4  = new swObject("agent4", agentSize,"method",methods,new
set(),INFINITY,"snmptrap");
   add(agent4);
   agent6  = new swObject("agent6", agentSize,"method",methods,new
set(),INFINITY,"snmptrap");
   add(agent6);
   agent15 = new swObject("agent15",agentSize,"method",methods,new
set(),INFINITY,"snmptrap");
   add(agent15);
   agent17 = new swObject("agent17",agentSize,"method",methods,new
set(),INFINITY,"snmptrap");
   add(agent17);
   agent19 = new swObject("agent19",agentSize,"method",methods,new
set(),INFINITY,"snmptrap");
   add(agent19);
}


public void Object_System_Mapping() {
   //couple DCO and LCN
   Add_coupling_swObject_to_processor( loop,    asc6  );
   Add_coupling_swObject_to_processor( mgr,     asc6  );
   Add_coupling_swObject_to_processor( agent4,  asc4  );
   Add_coupling_swObject_to_processor( agent6,  asc6  );
   Add_coupling_swObject_to_processor( agent15, asc15 );
   Add_coupling_swObject_to_processor( agent17, asc17 );
   Add_coupling_swObject_to_processor( agent19, asc19 );
}


public void Experimental_Frame() {
   //define DCO domain of study
   snmpSW = new set();
   snmpSW.add(loop);
   snmpSW.add(mgr);
   snmpSW.add(agent4);
   snmpSW.add(agent6);
   snmpSW.add(agent15);
   snmpSW.add(agent17);
   snmpSW.add(agent19);

   processors = new set();
   processors.add(asc4);
   processors.add(asc6);
   processors.add(asc15);
   processors.add(asc17);
   processors.add(asc19);
   //declare transducers
   String T = " Transducer";
   tAsc_snmp = new transd_domains(get_name()+" Domain"+T,snmpSW);
   add(tAsc_snmp);
   tTuple    = new transd_tuples("tuple"+T);
   add(tTuple);
```

```
   // declare acceptors
   String acceptorN = "acceptor";
   //set of: invocation msg: name,src,dst,msgSize,returnSize,msgType,
firingJob,calledMethod
   set invoke = new set();
   for (int i=0; i<loopCount; i++)
      invoke.add(new msg("invoke loop",acceptorN,loop.get_name(),
0,0,INFINITY,"invokeAsync",new job(),"loop"));
   //acceptor(Name,StartTime(sec),InvokeDutyCycle(sec),Repetitions,
InvokeMsgs,SimDutyCyle(sec),NumSimRuns
   acceptor = new acceptor(acceptorN,1,simTime,1,invoke,
simTime,simRuns);
   add(acceptor);

   //couple DCO and LCN to Acceptors and Transducers
   set asc6SW = new set();
   asc6SW.add(loop);
   asc6SW.add(mgr);
   asc6SW.add(agent6);

   Add_coupling_ethernetTransducer( link6, tLink6);

   Add_coupling_domainTransducer(    snmpSW, processors, tAsc_snmp );
   Add_coupling( tAsc_snmp,"results",   tTuple,"in");
   Add_coupling( tMsgs,    "results",   tTuple,"in");
   Add_coupling( tLink6,   "results",   tTuple,"in");
   // coupling for Acceptor
   Add_coupling( acceptor,"invoke",   loop,"inMsgs");
   Add_coupling_acceptorControl(acceptor, components);
}
}
```

B.2 Distributed Federation Simulation

```
/*
 *  Filename   : peFed.java
 *  Version    : 1.0
 *  Date       : 07-27-99
 *  Author     : Daryl Hild
*/

import DOC.*;
import java.lang.*;
import Zdevs.*;
import Zdevs.Zcontainer.*;

public class peFed extends digraphDCO {

   //simulation run parameters
   protected int    numPairs, simRuns, simulationIterations;
   protected double simTime;
   protected int    percentInteractionPairs, predictiveFilteringFactor;
   protected boolean multiplexedQuantizer;
   protected atomic ether;

public peFed(int numPrs, boolean muxQ, int filterFactor) {
   super("Pursuer-Evader_Federation");
   // 1 - no filtering; 5 - five-fold decrease in message traffic
   predictiveFilteringFactor = filterFactor;
   // multiplexed predictive quantization - "on" or "off"
   multiplexedQuantizer = muxQ;
   numPairs=numPrs;
   simulationIterations=100;
   simRuns=4;
   simTime=10000.0;
   int numProcs=2;
   peFed_construct(numProcs, numPairs);
}

public void peFed_construct(int numProcs, int numPairs) {
   //% of pursuer-evader pairs interacting in any simulation cycle
   percentInteractionPairs = 50;
   queue    processors = Loosely_Coupled_Network(numProcs);
   relation swObjects = Distributed_Cooperative_Objects(numPairs);
   Object_System_Mapping(swObjects, processors);
   Experimental_Frame(swObjects, processors);
   initialize();
}
```

```
public queue Loosely_Coupled_Network(int numProcessors) {
   //declare LCN components
   double rtiHS    = 20*8; //HLA RTI header size in bytes * 8 bits/byte
   double ipHS     = 20*8; // ip header size in bytes * 8 bits/byte
   double LS       = 2E+9; //proc nic speed bits/sec
   double BW       = INFINITY; //proc nic bandwidth bits/sec
   double BFinf    = INFINITY; //buffer size in bytes * 8 bits/byte
   double BF_hub   = 2E+6*8;   //buffer size in bytes * 8 bits/byte
   double BF_mau   = 2E+6*8;   //buffer size in bytes * 8 bits/byte
   double cpuSp    =  10E+3;   //cpu speed ops/sec
   double memSz    = 64E+6*8;  //cpu mem in bytes * 8 bits/byte
   double swapTime = 1.1;      //cpu-mem swapTimePenalty
   double ES       = 10E+6;    //ethernet speed bits/sec
   int    etherSeg = 5;        //number of Ethernet Segments
   int    MPS      = 1500;     //maximum packet size for LCN
   queue  procs    = new queue();

   ether = new link_ethernet("Ethernet",  etherSeg);   add(ether);

   for (int i=0; i<numProcessors; i++) {
      int j=i+1;
      digraph proc = new processor("proc_"+j,rtiHS+ipHS,LS,BW,
            BFinf,cpuSp,memSz,swapTime,MPS);
      add(proc);
      if (numProcessors>1)  {
         atomic mau = new hub_ethernet("mau"+j, ES, BW, BF_mau);
         add(mau);
         Add_coupling_LCNtoEthernet( proc,  mau,  ether);
      }
      procs.add(proc);
   }
   return procs;
}


public relation Distributed_Cooperative_Objects(int numPairs) {
   //declare DCO components
   dcoArc noArc = new dcoArc();

   set methods = new set();
   set arcs    = new set();

   //double timeOut = 4.0;
   double timeOut = INFINITY;

   int ACK   =  1*8; //  1 byte acknowledgement
   int RTIoh = 20*8; // 20 bytes RTI overhead
   int Sd    =  8*8; // attribute size for a "double" value
   int Si    =  2*8; // attribute size for an "int" value



   // fedex
```

```
int timeAdvGrant_wl =  20; // time advance grant workload
// methods:  methodName, queue_of_tasks
queue fedexCycle = new queue();
for (int i=0; i<simulationIterations; i++) {
   fedexCycle.add(new task(timeAdvGrant_wl,
      new dcoArc("timeAdvGrant(n)","rtiEx",RTIoh+Sd,ACK,
      "invokeSync","timeAdvGrant(n)",timeOut)));
   fedexCycle.add(new task(timeAdvGrant_wl,
      new dcoArc("timeAdvGrant(n+0.1)", "rtiEx",RTIoh+Sd,ACK,
      "invokeSync","timeAdvGrant(n+0.1)",timeOut)));
   fedexCycle.add(new task(timeAdvGrant_wl,
      new dcoArc("timeAdvGrant(n+0.2)","rtiEx",RTIoh+Sd,ACK,
      "invokeSync","timeAdvGrant(n+0.2)",timeOut)));
   fedexCycle.add(new task(timeAdvGrant_wl,
      new dcoArc("timeAdvGrant(n+0.3)", "rtiEx",RTIoh+Sd,ACK,
      "invokeSync","timeAdvGrant(n+0.3)",timeOut)));
}
methods = new set();
methods.add(new method("run()",   fedexCycle  ));
//swObject: name,size,threadMode,methods,arcs,dutyCycle,initMethod
atomic fedex = new swObject("fedex",64e+3*8,"none",methods,arcs,
      INFINITY,"run()");
add(fedex);


// rtiEx
int tick = RTIoh;
int tag_wl      =  10;
int tick_wl     =  10;
int send_wl     =  30;
int next_wl     =  20;
// methods:  methodName, queue_of_tasks
set federates = new set();
    federates.add(new entity("fed_P"));
    federates.add(new entity("fed_E"));
queue n00 = new queue();
      n00.add(new task( tag_wl,
               new dcoArc("timeAdvGrant(n)","devsCoord",RTIoh+Sd,
               tick,"invokeSync","timeAdvGrant(n)",timeOut)));
      n00.add(new task( tick_wl, noArc));
queue n01 = new queue();
      n01.add(new task( tag_wl,
               new dcoArc("timeAdvGrant(n+0.1)",federates,RTIoh+Sd,
               tick,"invokeSync","timeAdvGrant(n+0.1)",timeOut)));
      n01.add(new task( tick_wl, noArc));
queue n02 = new queue();
      n02.add(new task( tag_wl,
               new dcoArc("timeAdvGrant(n+0.2)",federates,RTIoh+Sd,
               tick,"invokeSync","timeAdvGrant(n+0.2)",timeOut)));
      n02.add(new task( tick_wl, noArc));
queue n03 = new queue();
```

```
      n03.add(new task( tag_wl,
              new dcoArc("timeAdvGrant(n+0.3)",federates,RTIoh+Sd,
              tick,"invokeSync","timeAdvGrant(n+0.3)",timeOut)));
      n03.add(new task( tick_wl, noArc));
queue send_GtN  =  new queue();
      send_GtN.add(new task( send_wl,
                   new dcoArc("send(Global_tN)",federates,RTIoh+Sd,
                   0,"message","send(Global_tN)",timeOut)));
queue send_iaP  =  new queue();
if (multiplexedQuantizer)
   send_iaP.add(new task( send_wl,
                new dcoArc("send(interactions)","fed_P",
                RTIoh+2*numPairs,0,"message","send(interactions)",tim
                eOut)));
else  for (int i=0; i<(numPairs*percentInteractionPairs)/
                        (100*predictiveFilteringFactor); i++)
   send_iaP.add(new task( send_wl,
                new dcoArc("send(interactions)","fed_P",
                RTIoh+Sd,0,"message","send(interactions)",
                timeOut)));
queue send_iaE  =  new queue();
if (multiplexedQuantizer)
   send_iaE.add(new task( send_wl,
                new dcoArc("send(interactions)","fed_E",
                RTIoh+2*numPairs,0,"message","send(interactions)",
                timeOut)));
else  for (int i=0; i<(numPairs*percentInteractionPairs)/
                        (100*predictiveFilteringFactor); i++)
   send_iaE.add(new task( send_wl,
                new dcoArc("send(interactions)","fed_E",
                RTIoh+Sd,0,"message","send(interactions)",
                timeOut)));
queue send_uaP  =  new queue();
if (multiplexedQuantizer)
   send_uaP.add(new task( send_wl,
                new dcoArc("send(updates)","fed_P",RTIoh+2*numPairs,
                0,"message","send(updates)", timeOut)));
else  for (int i=0; i<(numPairs*percentInteractionPairs)/
                        (100*predictiveFilteringFactor); i++)
   send_uaP.add(new task( send_wl,
                new dcoArc("send(updates)","fed_P",RTIoh+Sd,
                0,"message","send(updates)",timeOut)));
queue send_uaE  =  new queue();
if (multiplexedQuantizer)
   send_uaE.add(new task( send_wl,
                new dcoArc("send(updates)","fed_E",RTIoh+2*numPairs,
                0,"message","send(updates)",timeOut)));
else  for (int i=0; i<(numPairs*percentInteractionPairs)/
                        (100*predictiveFilteringFactor); i++)
   send_uaE.add(new task( send_wl,
                new dcoArc("send(updates)","fed_E",RTIoh+Sd,
                0,"message","send(updates)",timeOut)));
queue send_LtN  =  new queue();
```

```
      send_LtN.add(new task( send_wl,
                  new dcoArc("send(Local_tN)","devsCoord",RTIoh+Sd,
                  0,"message","send(Local_tN)",timeOut)));
queue nextEvRq  =  new queue();
      nextEvRq.add(new task( next_wl, noArc));
methods = new set();
methods.add(new method("timeAdvGrant(n)",       n00 ));
methods.add(new method("timeAdvGrant(n+0.1)",   n01 ));
methods.add(new method("timeAdvGrant(n+0.2)",   n02 ));
methods.add(new method("timeAdvGrant(n+0.3)",   n03 ));
methods.add(new method("send(Global_tN)",       send_GtN ));
methods.add(new method("send(interactionsToP)", send_iaP ));
methods.add(new method("send(interactionsToE)", send_iaE ));
methods.add(new method("send(updatesToP)",      send_uaP ));
methods.add(new method("send(updatesToE)",      send_uaE ));
methods.add(new method("send(Local_tN)",        send_LtN ));
methods.add(new method("nextEventReq()",        nextEvRq ));
//swObject: name,size,threadMode,methods,arcs,dutyCycle,initMethod
atomic rtiEx = new swObject("rtiEx",64e+3*8,"method",methods,arcs,
      INFINITY,"");
add(rtiEx);

// devsCoord
int send_GtN_wl =  50;
int nxtEvtRq_wl =  10;
int rcv_LtN_wl  =  20;
// methods:  methodName, queue_of_tasks
queue tag_n00  =  new queue();
      tag_n00.add(new task(send_GtN_wl,
                  new dcoArc("send(Global_tN)","rtiEx",RTIoh+Sd,
                  0,"message","send(Global_tN)",timeOut)));
      tag_n00.add(new task(nxtEvtRq_wl,
                  new dcoArc("nextEventReq()","rtiEx",RTIoh+Sd,
                  0,"message","nextEventReq()",timeOut)));
queue rcv_LtN  =  new queue();
      rcv_LtN.add(new task(rcv_LtN_wl, noArc));
methods = new set();
methods.add(new method("timeAdvGrant(n)", tag_n00));
methods.add(new method("send(Local_tN)",  rcv_LtN));
//swObject: name,size,threadMode,methods,arcs,dutyCycle,initMethod
atomic devsCoord = new swObject("devsCoord",64e+3*8,"method",
      methods,arcs,INFINITY,"");
add(devsCoord);

// DEVS federates
// fed_P and fed_E
int computeIO_wl =  10*numPairs;
int askallOUT_wl =  10*numPairs;
int whichDevs_wl = 200*numPairs;
int tellall_wl   =  10*numPairs;
int next_tN_wl   =  10*numPairs;
int interactQ_wl = 100*numPairs;
int updateQ_wl   = 100*numPairs;
```

```
int nxtEvtReq_wl =  10;
int rcv_GtN_wl   =  10;
int rcv_ia_wl    =  10;
int rcv_up_wl    =  10;
// fed_P
// methods:  methodName, queue_of_tasks
set Pursuers = new set();
    Pursuers.add(new entity("pursuers_"+numPairs));
int numP=numPairs; //model Pursuers as aggregate swObject
queue compute   = new queue();
      compute.add(new task(computeIO_wl,
            new dcoArc("compute_input_output(t)",Pursuers,numP*4*8,
            0,"invokeAsync","compute_input_output(t)",timeOut)));
      compute.add(new task(askallOUT_wl,
            new dcoArc("get_output()",Pursuers,numP*1*8, numP*40*8,
            "invokeSync","get_output()",timeOut)));
if (multiplexedQuantizer)
   compute.add(new task(whichDevs_wl,
           new dcoArc("send(interactionsToE)","rtiEx",
           RTIoh+2*numPairs,0,"message","send(interactionsToE)",
           timeOut)));
else  for (int i=0; i<(numPairs*percentInteractionPairs)/
                     (100*predictiveFilteringFactor); i++)
   compute.add(new task(whichDevs_wl,
           new dcoArc("send(interactionsToE)","rtiEx",  RTIoh+Sd,
           0,"message","send(interactionsToE)",timeOut)));
   compute.add(new task(nxtEvtReq_wl,
           new dcoArc("nextEventReq()","rtiEx",RTIoh+Sd,
           0,"message","nextEventReq()",timeOut)));
queue tellall = new queue();
      tellall.add(new task(whichDevs_wl, noArc));
      tellall.add(new task(tellall_wl,
            new dcoArc("wrap_deltfunc(tN,input)",Pursuers,numP*42*8,
            0,"invokeAsync","wrap_deltfunc(tN,input)",timeOut)));
if (multiplexedQuantizer)
      tellall.add(new task(whichDevs_wl,
            new dcoArc("send(updatesToE)","rtiEx",RTIoh+2*numPairs,
            0,"message","send(updatesToE)",timeOut)));
else  for (int i=0; i<(numPairs*percentInteractionPairs)/
                       (100*predictiveFilteringFactor); i++)
      tellall.add(new task(whichDevs_wl,
            new dcoArc("send(updatesToE)","rtiEx",RTIoh+Sd,
            0,"message","send(updatesToE)",timeOut)));
      tellall.add(new task(nxtEvtReq_wl,
            new dcoArc("nextEventReq()","rtiEx",RTIoh+Sd,
            0,"message","nextEventReq()",timeOut)));
queue next_tN = new queue();
      next_tN.add(new task(next_tN_wl,
            new dcoArc("next_tN()",Pursuers,numP*1*8,
            numP*4*8,"invokeSync","next_tN()",timeOut)));
      next_tN.add(new task(whichDevs_wl,
            new dcoArc("send(Local_tN)","rtiEx",RTIoh+Sd,
            0,"message","send(Local_tN)",timeOut)));
```

```
      next_tN.add(new task(nxtEvtReq_wl,
            new dcoArc("nextEventReq()","rtiEx",RTIoh+Sd,
            0,"message","nextEventReq()",timeOut)));
queue rcv_GtN  =  new queue();
      rcv_GtN.add(new task(rcv_GtN_wl, noArc));
queue rcv_ia   =  new queue();
      rcv_ia.add( new task(rcv_ia_wl, noArc));
queue rcv_up   =  new queue();
      rcv_up.add( new task(rcv_up_wl, noArc));
methods = new set();
methods.add(new method("timeAdvGrant(n+0.1)", compute ));
methods.add(new method("timeAdvGrant(n+0.2)", tellall ));
methods.add(new method("timeAdvGrant(n+0.3)", next_tN ));
methods.add(new method("send(Global_tN)",     rcv_GtN ));
methods.add(new method("send(interactions)",  rcv_ia  ));
methods.add(new method("send(updates)",       rcv_up  ));
// fed_P
//swObject: name,size,threadMode,methods,arcs,dutyCycle,initMethod
atomic fed_P = new swObject("fed_P",64e+3*8,"method",methods,
      arcs,INFINITY,"loop");
add(fed_P);
// fed_E
// methods:  methodName, queue_of_tasks
set Evaders  = new set();
Evaders.add(new entity("evaders_"+numPairs));
numP=numPairs; //model Evaders as aggregate swObject
compute   = new queue();
compute.add(new task(computeIO_wl,
            new dcoArc("compute_input_output(t)",Evaders,numP*4*8,
            0,"invokeAsync","compute_input_output(t)",timeOut)));
compute.add(new task(askallOUT_wl,
            new dcoArc("get_output()",Evaders,numP*1*8,
            numP*40*8,"invokeSync","get_output()",timeOut)));
compute.add(new task(nxtEvtReq_wl,
            new dcoArc("nextEventReq()","rtiEx",RTIoh+Sd,
            0,"message","nextEventReq()",timeOut)));
tellall = new queue();
tellall.add(new task(whichDevs_wl, noArc));
tellall.add(new task(tellall_wl,
            new dcoArc("wrap_deltfunc(tN,input)",Evaders,numP*42*8,
            0,"invokeAsync","wrap_deltfunc(tN,input)",timeOut)));
tellall.add(new task(nxtEvtReq_wl,
            new dcoArc("nextEventReq()","rtiEx",RTIoh+Sd,
            0,"message","nextEventReq()",timeOut)));
next_tN = new queue();
next_tN.add(new task(next_tN_wl,
            new dcoArc("next_tN()",Evaders,numP*1*8,
            numP*4*8,"invokeSync","next_tN()",timeOut)));
next_tN.add(new task(whichDevs_wl,
            new dcoArc("send(Local_tN)","rtiEx",RTIoh+Sd,
            0,"message","send(Local_tN)",timeOut)));
```

```
next_tN.add(new task(nxtEvtReq_wl,
            new dcoArc("nextEventReq()","rtiEx",RTIoh+Sd,
            0,"message","nextEventReq()",timeOut)));
methods = new set();
methods.add(new method("timeAdvGrant(n+0.1)", compute ));
methods.add(new method("timeAdvGrant(n+0.2)", tellall ));
methods.add(new method("timeAdvGrant(n+0.3)", next_tN ));
methods.add(new method("send(Global_tN)",     rcv_GtN ));
methods.add(new method("send(interactions)",  rcv_ia  ));
methods.add(new method("send(updates)",       rcv_up  ));
// fed_E
//swObject: name,size,threadMode,methods,arcs,dutyCycle,initMethod
atomic fed_E = new swObject("fed_E",64e+3*8,"method",methods,
        arcs,INFINITY,"loop");
add(fed_E);

// DEVS Models:  Pursuers and Evaders
int return_wl     =   1*numP; // return w/o processing    workload
int out_wl        = 100*numP; // out()                   workload
int get_output_wl =  10*numP; // get_output()            workload
int deltint_wl    = 200*numP; // deltint()               workload
int deltext_wl    = 300*numP; // deltext(e,x)            workload
// methods:  methodName, queue_of(pairs_of(workLoad, arc))
queue out = new queue();
      out.add( new task(75, return_wl, noArc));
      out.add( new task(25, out_wl,    noArc));
queue getOut = new queue();
      getOut.add(new task( get_output_wl, noArc));
queue wrap = new queue();
      wrap.add( new task( 60,return_wl, noArc));
      wrap.add( new task( 10,out_wl+deltint_wl+deltext_wl, noArc));
      wrap.add( new task( 10,out_wl+deltint_wl, noArc));
      wrap.add( new task( 10,deltext_wl, noArc));
queue next = new queue();
      next.add( new task( next_tN_wl, noArc));
methods = new set();
methods.add(new method("compute_input_output(t)",out   ));
methods.add(new method("get_output()",           getOut));
methods.add(new method("wrap_deltfunc(tN,input)",wrap  ));
methods.add(new method("next_tN()",              next  ));
//swObject: name,size,threadMode,methods,arcs,dutyCycle,initMethod
queue pursuers = new queue();
for (int i=0; i<numPairs; i++) {
   atomic pursuer;
   if (numP==1)
      pursuer = new swObject("pursuer_"+i,64e+3*8,"method",
            methods,arcs,INFINITY,"");
   else
      pursuer = new swObject("pursuers_"+numPairs,64e+3*8,"method",
            methods,arcs,INFINITY,"");
   add(pursuer);
   pursuers.add(pursuer);
   if (numP>1) break;
```

```
   }
   queue evaders = new queue();
   for (int i=0; i<numPairs; i++) {
      atomic evader;
      if (numP==1)
         evader = new swObject("evader_"+i,64e+3*8,"method",
               methods,arcs,INFINITY,"");
      else
         evader = new swObject("evaders_"+numPairs,64e+3*8,"method",
               methods,arcs,INFINITY,"");
      add(evader);
      evaders.add(evader);
      if (numP>1) break;
   }
   relation swObjects = new relation();
            swObjects.add(new entity("fedex"),       fedex);
            swObjects.add(new entity("rtiEx"),       rtiEx);
            swObjects.add(new entity("devsCoord"),   devsCoord);
            swObjects.add(new entity("fed_P"),       fed_P);
            swObjects.add(new entity("fed_E"),       fed_E);
            swObjects.add(new entity("pursuers"),    pursuers);
            swObjects.add(new entity("evaders"),     evaders);
   return swObjects;
}


public void Object_System_Mapping(relation swObjects, queue processors)
{
   //get processors
   digraph proc_1 = (digraph)processors.list_ref(0);
   digraph proc_2 = (digraph)processors.list_ref(1);
   //map fedex, rtiEx, devsCoord, fed_P, and pursuers to proc_1
   Add_coupling_swObject_to_processor( (atomic)
               swObjects.assoc("fedex"),     proc_1 );
   Add_coupling_swObject_to_processor( (atomic)
               swObjects.assoc("rtiEx"),     proc_1 );
   Add_coupling_swObject_to_processor( (atomic)
               swObjects.assoc("devsCoord"), proc_1 );
   Add_coupling_swObject_to_processor( (atomic)
               swObjects.assoc("fed_P"),     proc_1 );
   queue pursuers = (queue)swObjects.assoc("pursuers");
   int numPursuers = pursuers.get_length();
   for (int i=0; i<numPursuers; i++) {
      atomic pursuer = (atomic)pursuers.list_ref(i);
      Add_coupling_swObject_to_processor( pursuer, proc_1 );
   }
   //map fed_E & evaders to proc_2
   Add_coupling_swObject_to_processor( (atomic)
               swObjects.assoc("fed_E"),     proc_2 );
   queue evaders = (queue)swObjects.assoc("evaders");
   int numEvaders = evaders.get_length();
   for (int i=0; i<numEvaders; i++) {
      atomic evader = (atomic)evaders.list_ref(i);
```

```
      Add_coupling_swObject_to_processor( evader, proc_2 );
   }
}


public void Experimental_Frame(relation swObjects, queue processors) {
   //get processors
   digraph proc_1 = (digraph)processors.list_ref(0);
   digraph proc_2 = (digraph)processors.list_ref(1);
   atomic  fedex  = (atomic)swObjects.assoc("fedex");
   atomic  rtiEx  = (atomic)swObjects.assoc("rtiEx");

   set procs = new set();
       procs.add(proc_1);
       procs.add(proc_2);

   set proc_1_sw = new set();
       proc_1_sw.add(fedex);
       proc_1_sw.add(rtiEx);
       proc_1_sw.add(devsCoord);
       proc_1_sw.add(fed_P);
   for (int i=0; i<pursuers.get_length(); i++)
      proc_1_sw.add(pursuers.list_ref(i));
   set proc_2_sw = new set();
       proc_2_sw.add(fed_E);
   for (int i=0; i<evaders.get_length(); i++)
      proc_2_sw.add(evaders.list_ref(i));
   set fedDomain = proc_1_sw.union_objects(proc_2_sw);

   //declare transducers
   String T = " Transducer";
   atomic tRtiEx = new transd_swObj(rtiEx.get_name()+T,
           rtiEx.get_name());
   add(tRtiEx);
   atomic tEther = new transd_ethernet("Ethernet"+T);
   add(tEther);
   atomic tTuple = new transd_tuples("tuple"+T);
   add(tTuple);

   // declare acceptor
   String acceptorN = "acceptor";
   //set of msgs:name,src,dst,msgSize,rSize,msgType,fJob,calledMethod
   set invoke = new set();
   invoke.add(new msg("run()",acceptorN,fedex.get_name(),
           0,0,INFINITY,"invokeAsync",new job(),"run()"));
   atomic acceptor = new acceptor(acceptorN, 1, simTime, 1, invoke,
            simTime, simRuns);
   add(acceptor);

   //couple DCO and LCN to Acceptors and Transducers
   Add_coupling_msgsTransducer(      fedDomain,  procs,  tMsgs  );
   Add_coupling_swObjectTransducer(  rtiEx,      proc_1, tRtiEx );
   Add_coupling_ethernetTransducer(  ether,      tEther );
```

```
   Add_coupling( tEther,     "results",   tTuple,"in" );
   Add_coupling( tRtiEx,     "results",   tTuple,"in" );
   // coupling for Acceptor
   Add_coupling( acceptor,"invoke",   fedex,"inMsgs");
   Add_coupling_acceptorControl(acceptor, components);
}
}
```

APPENDIX C. DEVS-DOC BEHAVIOR SPECIFICATIONS

In this appendix, dynamic behavior specifications for the DEVS-DOC components are provided using the "Parallel DEVS with Ports" formalism.

C.1 LCN: link_ethernet

$DEVS_{link\_ethernet} = < X, Y, S, \delta_{int}, \delta_{ext}, \delta_{conf}, \lambda, ta >$, where

InPorts = {"in"}
OutPorts = {"out"}
X = {(f,r) | f is arbitrary, $r \in \Re$}
Y = {preamble, f_{last}, $f_{\varnothing}$}
where $x_{last}=(f_{last},r_{last})$ represents the last input pair received
$S = \{\text{"passive","xmitting","collisions","noiseburst"}\} \times \Re_0^+ \times x_{last}$

$\delta_{ext}((phase,\sigma,x_{last}),e,(\text{"in"},x)) =$ case phase is

("preamble",propagtionTime,x)	"passive"
("collisions",σ-e,x)	"preamble"
("collisions",σ-e,x)	"xmitting"
("collisions",σ-e,x)	"collisions"
("noiseburst",propagationTime,x)	"noiseburst"

$\delta_{int}(phase,\sigma,x_{last}) =$ case phase is

("passive",∞,($\varnothing$,$\varnothing$))	"passive"
("xmitting",r_{last},x_{last})	"preamble"
("passive",∞,($\varnothing$,$\varnothing$))	"xmitting"
("noiseburst",∞,($\varnothing$,$\varnothing$))	"collisions"
("passive",∞,($\varnothing$,$\varnothing$))	"noiseburst"

$\delta_{conf}(s,ta(s),x) = \delta_{ext}(\delta_{int}(s),0,x)$

$\lambda\ (phase,\sigma,x_{last}) =$ case phase is

("out",preamble)	"preamble"
("out",f_{last})	"xmitting"
("out",$f_{\varnothing}$)	"collisions"
("out",noiseburst)	"noiseburst"

$ta(phase,\sigma,x_{last}) = \sigma$

C.2 LCN: hub_ethernet

To simplify the specification, only one ethernet port is assumed.

$DEVS_{hub\_ethernet}$ = < X, Y, S, δ_{int}, δ_{ext}, δ_{conf}, λ, ta >, where

InPorts = {inLoop, in1}
OutPorts = {out Loop, out1}
X = { (inPort, pdu) }, where pdu = (source, dest, size, data)
Y = { (outLoop, pdu) }
{ (out1, (pdu,r)) | r ∈ $\Re$ }
S = Phase × σ × XmitState × MediaState × LoopDelay × LoopBuffer
× PortDelay × PortBuffer × BackOffCount
Phase = {passive,busy}
σ = $\Re_0^+$
XmitState = {idle, waitingForIdle, xmitting}
MediaState = {idle, singleCarrier, collisions}
LoopDelay = $\Re_0^+$
LoopBuffer = L^+ (a FIFO queue of pdu's)
PortDelay = $\Re_0^+$
PortBuffer = P^+ (a FIFO queue of pairs of pdu's and xmitTimes)
BackOffCount = an integer ≥ 0

δ_{ext}(s,*e*,(InPorts,X)) =

(,,,,loopDelay-*e*,,portDelay-*e*,,) before processing input events X

(,,,,,,,P^+.add(x,xt),) for each x event on "inLoop"
where xt is xmitTime for x, i.e., xt=x.pdu_size/ethernetSpeed

(,,xs,ms,ld,lb,pd,pb,boc) for each x event on "in"
where ms is new mediaState,
if x.pdu=$f_\varnothing$ ms=collisions
else if x.pdu=preamble ms=singleCarrier
else if x.pdu=noiseburst ms=idle
else ms=idle
where xs is new xmitState,
if x.pdu!= $f_\varnothing$ and x.pdu!=preamble and x.pdu!=noiseburst
and xmitState=xmitting
if P^+.size=1 xs=idle
else xs=waitingForIdle

```
    where boc is new BackOffCount,
            if x.pdu!= f∅ and x.pdu!=preamble and x.pdu!=noiseburst
              and xmitState=xmitting
                    if P+.size=1      boc=0
                    else              boc=BackOffCount++
    where pb is new portBuffer,
            if x.pdu!= f∅ and x.pdu!=preamble and x.pdu!=noiseburst
              and xmitState=xmitting              pb= P+.removeFrontPair
    where pd is new portDelay,
            if x.pdu!= f∅ and x.pdu!=preamble and x.pdu!=noiseburst
              and xmitState=xmitting
                    if P+.size=1      pd=∞
                    else              pd=RandomInt[0..2^BackOffCount++-1]
    where lb is new loopBuffer,
            if x.pdu!= f∅ and x.pdu!=preamble and x.pdu!=noiseburst
              and xmitState!=xmitting             lb=L+.add(x.pdu)
    where ld is new loopDelay,
            if x.pdu!= f∅ and x.pdu!=preamble and x.pdu!=noiseburst
              and xmitState!=xmitting
                    if loopDelay=∞                ld= x.pdu_size/loopSpeed

(ph,sigma,xs,,,,pd,,boc)                  after processing input events X
    where xs is new xmitState,
            if xmitState=idle  and P+.size>0       xs=waitingForIdle
    where pd is new portDelay,
            if xmitState=idle  and P+.size>0  and  mediaState=idle
                                                  pd=0.0
            else if xmitState=idle  and P+.size>0  and  mediaState!=idle
                                      pd=RandomInt[0..2^BackOffCount++-1]
            else if xmitState=waitingForIdle  and portDelay=∞
                                      pd=RandomInt[0..2^BackOffCount++-1]
            else if xmitState=xmitting  and  mediaState=collisions
                                                  pd=0.0
    where boc is new backOffCount,
            if xmitState=idle  and P+.size>0  and  mediaState!=idle
                                                  boc=BackOffCount++
            else if xmitState=waitingForIdle  and portDelay=∞
                                                  boc=BackOffCount++
```

where ph is new phase,
    if minimum(loopDelay,pd)=∞ ph=passive
    else ph=busy
where sigma is new σ,
    sigma=minimum(loopDelay,pd)

$\delta_{int}(s)$ = (ph,sigma,xs,,ld,lb,pd,,boc)
where ld is new loopDelay
    if loopDelay≤σ and L^{+}.size=1
        ld=∞
    else if loopDelay≤σ ld=L^{+}.next_pdu.size/loopSpeed
    else ld=loopDelay-σ
where lb is new loopBuffer
    if loopDelay≤σ lb=L^{+}.removeFront_pdu
where xs is new xmitState
    if portDelay≤σ and xmitState=waitingForIdle
      and mediaState=xmitting
        xs=xmitting
    if portDelay≤σ and xmitState=xmitting
      and mediaState=collisions
        xs=waitingForIdle
where pd is new portDelay
    if portDelay≤σ and xmitState=waitingForIdle
      and mediaState=xmitting
        pd=∞
    else if portDelay≤σ and xmitState=waitingForIdle
        pd=RandomInt[0..2^BackOffCount++-1]
    else if portDelay≤σ and xmitState=xmitting
      and mediaState=collisions
        pd=∞
    else pd=portDelay-σ
where boc is new backOffCount
    if portDelay≤σ and xmitState=waitingForIdle
        boc=backOffCount++
where ph is new phase,
    if ld=∞ and pd=∞ ph=passive
    else ph=busy
where sigma is new σ,
    if ld=∞ and pd=∞ sigma=∞
    else sigma=minimum(ld,pd)

$\delta_{conf}(s,ta(s),x) = \delta_{ext}(\delta_{int}(s),0,x)$

$\lambda(s) = y^{+}$

where y^{+} is a set of output (port, value) pairs

y^{+} = new empty set

if loopDelay≤σ

y^{+}.add(outLoop,L^{+}.front_pdu)

if portDelay≤σ and xmitState=waitingForIdle

and mediaState=idle

y^{+}.add(out1,P^{+}.front_pair)

else if portDelay≤σ and xmitState=xmitting

and mediaState=collisions

y^{+}.add(out1,(noiseburst,0.0))

$ta(s) = \sigma$

C.3 LCN: router

To simplify the specification, only two router ports are assumed.

$DEVS_{router}$ = < X, Y, S, δ_{int}, δ_{ext}, δ_{conf}, λ, ta >, where

InPorts = {inLoop, inLink1, inLink2}

OutPorts = {outLoop, outLink1, outLink2}

X = { (inPort, pdu) }, where pdu = (source, dest, size, data)

Y = { (outPort, pdu) }

S = Phase × σ × OutLoopBuffer × Out1Buffer × Out2Buffer
× OutLoopDelay × Out1Delay × Out2Delay × AddressList

Phase = {passive,busy}

$\sigma = \Re_0^+$

OutLoopBuffer = L^+ (a FIFO queue of pdu's)

OutBuffer1 = $O1^+$ (a FIFO queue of pdu's)

OutBuffer2 = $O2^+$ (a FIFO queue of pdu's)

LoopDelay = $\Re_0^+$

OutDelay1 = $\Re_0^+$

OutDelay2 = $\Re_0^+$

AddressList = A^+ (a function of software names to output ports)

δ_{ext}(s,*e*,(InPorts,X)) =

(,,,,,ld,od1,od2,) — before processing input events X

where ld is new LoopDelay, — ld=LoopDelay-*e*

where od1 is new OutDelay1, — o1d=OutDelay1-*e*

where od2 is new OutDelay2, — o2d=OutDelay2-*e*

(,,,ob1,ob2,,od1,od2,al) — for each x event on "inLoop"

where ob1 is new OutBuffer1,

if x.destination in AddressList
and AddressList.association(x.destination)=outLink1
ob1=$O1^+$.add(x)

else if x.destination not in AddressList
and x.data=load
ob1=$O1^+$.add(x)

where ob2 is new OutBuffer2,

if x.destination in AddressList
and AddressList.association(x.destination)=outLink2
ob2=$O2^+$.add(x)

else if x.destination not in AddressList
and x.data=load

```
                                        ob2=O2+.add(x)
    where od1 is new OutDelay1,
          if x.destination in AddressList
            and AddressList.association(x.destination)=outLink1
                                        od1=x.size/outLink1Speed
          else if x.destination not in AddressList
            and x.data=load
                                        od1=x.size/outLink1Speed
    where od2 is new OutDelay2,
          if x.destination in AddressList
            and AddressList.association(x.destination)=outLink2
                                        od2=x.size/outLink2Speed
          else if x.destination not in AddressList
            and x.data=load
                                        od2=x.size/outLink2Speed
    where al is new AddressList,
          if x.destination not in AddressList
            and x.data=load
                                        al= A+.add(x.destination,outLoop)

(,,lb,,ob2,ld,,od2,al)                  for each x event on "inLink1"
    where lb is new OutLoopBuffer,
          if x.destination in AddressList
            and AddressList.association(x.destination)=outLoop
                                        lb=L1+.add(x)
          else if x.destination not in AddressList
            and x.data=load
                                        lb=L1+.add(x)
    where ob2 is new OutBuffer2,
          if x.destination in AddressList
            and AddressList.association(x.destination)=OutLink2
                                        ob2=O2+.add(x)
          else if x.destination not in AddressList
            and x.data=load
                                        ob2=O2+.add(x)
    where ld is new LoopDelay,
          if x.destination in AddressList
            and AddressList.association(x.destination)=outLoop
                                        ld=x.size/LoopSpeed
          else if x.destination not in AddressList
            and x.data=load
                                        ld=x.size/LoopSpeed
    where od2 is new OutDelay2,
```

```
            if x.destination in AddressList
              and AddressList.association(x.destination)=outLink2
                                        od2=x.size/outLink2Speed
            else if x.destination not in AddressList
              and x.data=load
                                        od2=x.size/outLink2Speed
        where al is new AddressList,
            if x.destination not in AddressList
              and x.data=load
                                        al= A+.add(x.destination,outLink1)

(,,lb,ob1,,ld,od1,,al)                  for each x event on "inLink2"
        where lb is new OutLoopBuffer,
            if x.destination in AddressList
              and AddressList.association(x.destination)=outLoop
                                        lb=L1+.add(x)
            else if x.destination not in AddressList
              and x.data=load
                                        lb=L1+.add(x)
        where ob1 is new OutBuffer1,
            if x.destination in AddressList
              and AddressList.association(x.destination)=outLink1
                                        ob1=O1+.add(x)
            else if x.destination not in AddressList
              and x.data=load
                                        ob1=O1+.add(x)
        where ld is new LoopDelay,
            if x.destination in AddressList
              and AddressList.association(x.destination)=outLoop
                                        ld=x.size/LoopSpeed
            else if x.destination not in AddressList
              and x.data=load
                                        ld=x.size/LoopSpeed
        where od1 is new OutDelay1,
            if x.destination in AddressList
              and AddressList.association(x.destination)=outLink1
                                        od1=x.size/outLink1Speed
            else if x.destination not in AddressList
              and x.data=load
                                        od1=x.size/outLink1Speed
        where al is new AddressList,
            if x.destination not in AddressList
              and x.data=load
```

al= A^{+}.add(x.destination,outLink2)

(ph,sigma,,,,,,,) after processing input events X

where ph is new phase,

if minimum(LoopDelay,OutDelay1,OutDelay2)=∞ ph=passive

else ph=busy

where sigma is new σ,

sigma=minimum(loopDelay,pd)

δ_{int}(s) = (ph,sigma,lb,ob1,ob2,ld,od1,od2,)

where ld is new LoopDelay, ld=LoopDelay-σ

where od1 is new OutDelay1, od1=OutDelay1-σ

where od2 is new OutDelay2, od2=OutDelay2-σ

where lb is new LoopBuffer,

if ld≤0 lb=L^{+}.removeFront_pdu

if lb is empty ld=∞

else ld=lb.front_pdu.size/outLoopSpeed

where ob1 is new OutBuffer1,

if od1≤0 ob1=$O1^{+}$.removeFront_pdu

if ob1 is empty od1=∞

else od1=ob1.front_pdu.size/outLink1Speed

where ob2 is new OutBuffer2,

if od2≤0 ob2=$O2^{+}$.removeFront_pdu

if ob2 is empty od2=∞

else od2=ob2.front_pdu.size/outLink2Speed

where ph is new phase,

if minimum(ld,od1,od2)<∞ ph=busy

else ph=passive

where sigma is new σ,

sigma=minimum(ld,od1,od2)

δ_{conf}(s,ta(s),x) = δ_{ext}(δ_{int}(s),0,x)

λ (s) = y^{+}

where y^{+} is a set of output (port, value) pairs

y^{+} = new empty set

if LoopDelay≤σ y^{+}.add(outLoop, L^{+}.front_pdu)

else if OutDelay1≤σ y^{+}.add(outLink1, $O1^{+}$.front_pdu)

else if OutDelay2≤σ y^{+}.add(outLink2, $O2^{+}$.front_pdu)

ta(s) = σ

C.4 LCN: cpu_singleTask

$DEVS_{cpu\_singleTask}$ = < X, Y, S, δ_{int}, δ_{ext}, δ_{conf}, λ, ta >, where

InPorts = {inJobs, inSW}

OutPorts = {outJobs}

X = { (inJobs, job) }, where job = (swObject, workLoad)
{ (inSW, pdu) }, where pdu = (source, dest, size, data)

Y = { (outJobs, job) }

S = Phase $\times$ σ $\times$ memSW $\times$ Jobs

Phase = {passive,busy}

$\sigma = \Re_0^+$

memSW = a set of swObject names

Jobs = a relation of jobTime to job

δ_{ext}(s,*e*,(InPorts,X)) =

(,,mSW,) for each x event on "inSW"
where mSW is new memSW
if x.dest=mem and x.data=load
mSW=memSW.add(x.source)
else if x.dest=mem and x.data=unload
mSW=memSW.remove(x.source)

(,,,Js) for each x event on "inJobs"
where Js is new Jobs
if x.src is in memSW Js=Jobs.add(x)

(ph,sigma,,) process last
where ph is new phase
if Jobs is empty ph=passive
else ph=busy
where sigma is new σ
if Jobs is empty sigma=∞
else if phase=busy sigma=σ-*e*
else sigma=Jobs.front_job.size/cpuSpeed

δ_{int}(s) = (ph,sigma,,Js)
where ph is new phase
if Jobs.size=1 ph=passive
else ph=busy
where sigma is new σ
if Jobs.size=1 sigma=∞

else sigma=Jobs.next_job.size/cpuSpeed
where Js is new Jobs
Js=Jobs.remove_front_pdu

$\delta_{conf}(s,ta(s),x) = \delta_{ext}(\delta_{int}(s),0,x)$

λ (s) = ("outJobs",Jobs.front_job) if Jobs not empty

ta(s) = σ

C.5 LCN: cpu_multiTask

$DEVS_{cpu\_multiTask} = < X, Y, S, \delta_{int}, \delta_{ext}, \delta_{conf}, \lambda, ta >$, where

InPorts = {inJobs, inSW}

OutPorts = {outJobs}

X = { (inJobs, job) }, where job = (swObject, workLoad)
{ (inSW, pdu) }, where pdu = (source, dest, size, data)

Y = { (outJobs, job) }

S = Phase × σ × memSW × Jobs × memInUse

Phase = {passive,busy}

$\sigma = \Re_0^+$

memSW = a set of swObject names

Jobs = a relation of jobTime to job

memInUse = $\Re_0^+$

$\delta_{ext}(s,e,(InPorts,X)) =$

```
(,,mSW,,mIU)                                    for each x event on "inSW"
    where mSW is new memSW
            if  x.dest=mem  and  x.data=load
                                                mSW=memSW.add(x.source)
            else if  x.dest=mem  and  x.data=unload
                                                mSW=memSW.remove(x.source)
    where mIU is new memInUse
            if  x.dest=mem  and  x.data=load
                                                mIU=memInUse+x.size
            else if  x.dest=mem  and  x.data=unload
                                                mIU=memInUse-x.size

oldJobCount=Jobs.length
(,,,Js,)                                        for each x event on "inJobs"
    where Js is new Jobs
            if  x.src is in memSW
                    newJobTime=(e+x.workLoad/cpuSpeed)*oldJobCount
                    Js=Jobs.add(newJobTime,x)

newJobCount=Jobs.length
(ph,sigma,Js,,)                         for each (jobTime_i, job_i) pair in Jobs
    where Js is new Jobs
            newJobTime_i=(jobTime_i-e)newJobCount/oldJobCount
            Js=Js.add(newJobTime_i,job_i)
    where ph is new phase
            if Jobs is empty            ph=passive
```

```
            else                    ph=busy
        where sigma is new σ
            if Jobs is empty        sigma=∞
            else                    sigma= minimum(each jobTime_i in Jobs)
```

$\delta_{int}(s)$ =

```
    oldJobCount=Jobs.length
    (,,,Js,)                        for each (jobTime_i, job_i) pair in Jobs
        where Js is new Jobs
            if jobTime_i≤σ          Js= J+.remove_pair(jobTime_i, job_i)

    newJobCount=Jobs.length
    (ph,sigma,,Js,)                 for each (jobTime_i, job_i) pair in Jobs
        where Js is new Jobs
            newJobTime_i=(jobTime_i-sigma)newJobCount/oldJobCount
            Js=Js.add(newJobTime_i,job_i)
        where ph is new phase
            if Jobs is empty        ph=passive
            else                    ph=busy
        where sigma is new σ
            if Jobs is empty        sigma=∞
            else                    sigma= minimum(each jobTime_i in Jobs)
```

$\delta_{conf}(s,ta(s),x) = \delta_{ext}(\delta_{int}(s),0,x)$

$\lambda(s) = y$

```
        where y is a set of output (port, value) pairs
            y = new empty set
            for each (jobTime, job) pair in Jobs
                if      jobTime≤σ       y.add(outJob, job)
```

$ta(s) = \sigma$

C.6 LCN: transport

$DEVS_{transport}$ = < X, Y, S, δ_{int}, δ_{ext}, δ_{conf}, λ, ta >, where

InPorts = {inMsgs, inPkts}

OutPorts = {outMsgs, outPkts}

X = { (inMsgs, pdu) , (inPkts, pdu) }, where pdu = (source, dest, size, data)

Y = { (outMsgs, pdu), (outPkts, pdu) }

S = Phase × σ × xmittingQ × deliveryQ × recvingQ

Phase = {passive,busy}

$\sigma = \Re_0^+$

xmittingQ = a set of msg, where msg=(src,dest,size,data)

deliveryQ = a set of msg

recvingQ = a function of (msg → pktQ) pairs,

where pktQ is queue of pkt and pkt=(src,dest,size,data)

δ_{ext}(s,*e*,(InPorts,X)) =

(ph,sigma,xQ,,) for each x event on "inMsgs"

where ph is new phase, ph=busy

where sigma is new σ, sigma=0

where xQ is new xmittingQ, xQ=xmittingQ.add(x)

(ph,sigma,,dQ,rQ) for each x event on "inPkts"

m=x.msg

pQ=recvingQ(m)

pQ.add(x)

where ph is new phase, ph=busy

where sigma is new σ,

if x.num_fragments=pQ.length sigma=0

where dQ is new deliveryQ,

if x.num_fragments=pQ.length dQ=deliveryQ.add(m)

where rQ is new recvingQ,

if x.num_fragments=pQ.length rQ=recvingQ.remove(m→pQ)

else rQ=recvingQ.replace(m→pQ)

δ_{int}(s) = (ph,sigma,xQ,dQ,rQ)

where ph is new phase

if recvingQ is empty ph=passive

else ph=busy

where sigma is new σ, sigma=∞

where xQ is new xmittingQ, xQ=empty set

where dQ is new deliveryQ, dQ=empty set

δ_{conf}(s,ta(s),x) = δ_{ext}(δ_{int}(s),0,x)

λ (s) = y
where y is a set of output (port, value) pairs
y = new empty set
for each msg in xmittingQ,
partition msg into pkts of size maxPktSize
for each pkt, y=y.add("outPkts",pkt)
for each msg in deliveryQ,
y=y.add("outMsgs",msg)

ta(s) = σ

C.7 DCO: swObject

$DEVS_{swObject}$ = < X, Y, S, δ_{int}, δ_{ext}, δ_{conf}, λ, ta >, where

InPorts = {inMsgs, doneJobs}

OutPorts = {outMsgs, outJobs, outSW}

X = { (inMsgs, msg) , (doneJobs, job) },

where msg=(name,src,dest,size,returnSize,timeOut,msgType,firingJob,method)

where job = (swObject, method, workLoad, workLeft, invokingMessage)

Y = { (outMsgs, msg), (outJobs, job), (outSW, pdu) }

where pdu = (src, dest, size, data)

S = Phase × σ × activeJobs × commJobs × queuedJobs × timerMsgs
× fireJobs × fireMsgs × loadStatus

where

Phase = {passive, active, fire}

$\sigma = \Re_0^+$

activeJobs = relation of (method→job)

commJobs = set of jobs

queuedJobs = relation of (method→job)

timerMsgs = relation of (timeOut→msg)

fireJobs = set of jobs

fireMsgs = set of msgs

loadStatus = {unloaded, onDisk, inMem, unloadMem, unloadDisk}

δ_{ext}(s,*e*,(InPorts,X)) =

(ph, sigma, aj, cj, qj, tm, fj, fm, ls) for each inJob on "doneJobs"
and inJob.swObject=self

where ph is new phase
sigma is new σ
aj is new activeJobs, aj=activeJobs
cj is new commJobs, cj=commJobs
qj is new queuedJobs, qj=queuedJobs
tm is new timerMsgs, tm=timerMsgs
fj is new fireJobs, fj=fireJobs
fm is new fireMsgs, fm=fireMsgs
ls is new loadStatus, ls=loadStatus
inJob.reset_workLoad
if inJob.method=""
for each arc_i in arcs
if arc_i.workLoad≤inJob.workDone
fm=fm.add(newMsg base on arc_i)
ph=fire
sigma=0

```
                if  arc_i.returnSize>0
                        cj=cj.add(inJob)
                        if  arc_i.invokeSynchronous   inJob.set_to_blocked
                        if arc_i.timeOut>0  tm=tm.add(timeOut→newMsg)
                arc_i.set_workLoad(arc_i.totalWorkLoad)
          else   arc_i.set_workLoad(workLoad-inJob.workDone)
else
     get next task_i in task based on inJobs.workDone
     for each arc_i in task_i
                fm=fm.add(newMsg base on arc_i)
                ph=fire
                sigma=0
                if  arc_i.returnSize>0
                        cj=cj.add(inJob)
                        if  arc_i.invokeSynchronous   inJob.set_to_blocked
                        if arc_i.timeOut>0  tm=tm.add(timeOut→newMsg)
if  inJob.is_not_blocked
     if inJob.workLoad>0  fj=fj.add(inJob)
                          ph=fire
                          sigma=0
     else
                aj=aj.remove(inJob.method,inJob)
                if  threadMode=none
                    if   qj is not empty
                        nextJob=qj.front_job
                        qj=qj.remove_front_job
                        aj=aj.add(nextJob.method→nextJob)
                        fj=fj.add(nextJob)
                        ph=fire
                        sigma=0
                     else if  cj is empty
                        ls=unloadMem
                        ph=fire
                        sigma=0
                if  threadMode=object
                   if  qj is empty   and   aj is empty   and    cj is empty
                        ls=unloadMem
                        ph=fire
                        sigma=0
                   else if  qj is not empty
                        nextJob=qj.associated(inJob.method)
                        if  nextJob exists
                                qj.remove(inJob.method→nextJob)
```

```
                                    aj=aj.add(nextJob.method→nextJob)
                                    fj=fj.add(nextJob)
                                    ph=fire
                                    sigma=0
                        if  threadMode=method
                           if   aj is empty   and   cj is empty
                                ph=fire
                                sigma=0
                                ls=unloadMem
                        if  inJob.invokeMsg.returnSize>0
                                fireMsgs(inJob.invokeMsg.returnMsg)
                                ph=fire
                                sigma=0


(ph, sigma, aj, cj, qj, tm, fj, fm, ls)          for each inMsg on "inMsgs"
                                                        and  inMsg.dest=self
        where  ph is new phase
               sigma is new σ
               aj is new activeJobs,             aj=activeJobs
               cj is new commJobs,               cj=commJobs
               qj is new queuedJobs,             qj=queuedJobs
               tm is new timerMsgs,              tm=timerMsgs
               fj is new fireJobs,               fj=fireJobs
               fm is new fireMsgs,               fm=fireMsgs
               ls is new loadStatus,             ls=loadStatus
            if  inMsg.isReturnMsg
              if  inMsg.firingJob is in commJobs
                tm=tm.remove(*→inMsg)
                if  inMsg.firingJob.is_blocked
                   firingJob.set_unblocked
                   cj=cj.remove(firingJob)
                   if  firingJob.workLeft>0
                         fj=fj.add(firingJob)
                         ph=fire
                         sigma=0
                   else
                         aj=aj.remove(inJob.method,inJob)
                         if  threadMode=none
                            if   qj is not empty
                                 nextJob=qj.front_job
                                 qj=qj.remove_front_job
                                 aj=aj.add(nextJob.method→nextJob)
```

```
                        fj=fj.add(nextJob)
                        ph=fire
                        sigma=0
                    else if  cj is empty
                        ls=unloadMem
                        ph=fire
                        sigma=0
                if  threadMode=object
                    if  qj is empty   and   aj is empty   and    cj is empty
                        ls=unloadMem
                        ph=fire
                        sigma=0
                    else if  qj is not empty
                        nextJob=qj.associated(inJob.method)
                        if  nextJob exists
                            qj.remove(inJob.method→nextJob)
                            aj=aj.add(nextJob.method→nextJob)
                            fj=fj.add(nextJob)
                            ph=fire
                            sigma=0
                if  threadMode=method
                    if   aj is empty   and   cj is empty
                        ph=fire
                        sigma=0
                        ls=unloadMem
                if  inJob.invokeMsg.returnSize>0
                        fireMsgs(inJob.invokeMsg.returnMsg)
                        ph=fire
                        sigma=0
            else
                if  phase=passive       aj=aj.add(newJob based on inMsg)
```

$\delta_{int}(s)$ = (ph, sigma, aj, cj, qj, tm, fj, fm, ls)

```
  where ph is new phase
        if  phase=passive               ph=fire
        else if  loadStatus=inMem       ph=active
        else                            ph=passive
  where sigma is new σ
        if  phase=passive               sigma=0
        else if  loadStatus=inMem       sigma=min(each timeOut_i in timerMsgs)
        else                            sigma=∞
  where aj is new activeJobs
        if  phase=passive       aj={newJob based on initialization_message}
```

```
        else                        aj=activeJobs
  where cj is new commJobs          cj=commJobs
  where qj is new queuedJobs        qj=queuedJobs
  where tm is new timerMsgs
        for each (timeOut_i, msg_i) in timerMsgs
              if  timeOut_i≤σ       tm=tm.add(msg_i.timeOut, msg_i)
              else                  tm=tm.add(timeOut_i-σ, msg_i)
  where fj is new fireJobs
        if  phase=passive           fj=aj.newJob
        else                        fj=empty set
  where fm is new fireMsgs          fm=empty set
  where ls is new loadStatus
        if  loadStatus=unloaded                          ls=onDisk
        else if  loadStatus=onDisk
          and  (fireJobs.length+fireMsgs.length)>0       ls=inMem
        else if  loadStatus=unloadMem                    ls=onDisk
        else if  loadStatus=unloadDisk                   ls=unloaded
```

$\delta_{conf}(s,ta(s),x) = \delta_{ext}(\delta_{int}(s),0,x)$

$\lambda\ (s) = y$

```
  where y is a set of output (port, value) pairs
        y = new empty set
        if  loadStatus=unloaded
              y=y.add((outSW,(myName,Disk,swSize,load))
        else if  loadStatus=onDisk  and  (fireJobs.length + fireMsgs.length)>0
              y=y.add((outSW,(myName,Mem,swSize,load))
        else if  loadStatus=unloadMem  and  fireJobs.length=0
              y=y.add((outSW,(myName,Mem,swSize,unload))
        else if  loadStatus=unloadDisk
              y=y.add((outSW,(myName,Disk,swSize,unload))
        for each job_i in fireJobs
              y=y.add("outJobs",job_i)
        for each msg _i in fireMsgs
              y=y.add("outMsgs",msg_i)
        for each (timeOut_i, msg _i) in timerMsgs
              if  timeOut_i=σ       y=y.add("outMsgs",msg_i)
```

$ta(s) = \sigma$

C.8 Experimental Frame: acceptor

$DEVS_{acceptor}$= < X, Y, S, δ_{int}, δ_{ext}, δ_{conf}, λ, ta >, where

InPorts = { control }

OutPorts = { control, invoke }

X = { (control, controlMsg) }

where controlMsg = {passivate, initialize}

Y = { (control, controlMsg), (invoke, msg) }

where msg=(name,src,dest,size,returnSize,timeOut,msgType,firingJob,method)

S = Phase × σ × ControlMsg × NumSimRuns × Repetition

Phase = {passive, initializing, starting, collecting, stopping, reporting}

$\sigma = \Re_0^+$

ControlMsg = {∅, passivate, initialize}

$NumSimRuns = \Im_0^+$

$Repetition = \Im_0^+$

δ_{ext}(s,*e*,(InPorts,X)) =

(,sigma,)	before processing events
where sigma is new σ,	sigma=σ-*e*
(ph,sigma,cM)	for each x event on "control"
where ph is new Phase,	ph=Phase
where sigma is new σ,	sigma=0
where cM is new ControlMsg,	cM=x

δ_{int}(s) = (ph,sigma,cM,nSR,rep)

where cM is new ControlMsg,		cM=∅
where ph is new Phase		
sigma is new σ,		
nSR is new NumSimRuns		
rep is new Repetition		
	if Phase=initializing	ph=starting
		sigma=startupTime
		nSR=numSimRuns-1
	else if Phase=starting	ph=collecting
		sigma=invokeDutyCycle
		rep=repetitions - 1;

else if Phase=collecting and repetition≥0	ph=collecting sigma=invokeDutyCycle rep=repetition--;
else if Phase=collecting and repetition=0	ph=stopping sigma=simDutyCycle rep=repetition--;
else if Phase=stopping	ph=reporting sigma=0
else if Phase=reporting and numSimRuns>1	ph=initializing sigma=1
else if Phase=reporting and numSimRuns=1	ph=passivate sigma=∞
else	ph=passivate sigma=∞

$\delta_{conf}(s,ta(s),x) = \delta_{ext}(\delta_{int}(s),0,x)$

$\lambda(s) = y$

where y is a set of output (port, value) pairs

```
y = new empty set
if  ControlMsg!=∅                y=y.add(control, ControlMsg)
if  Phase=initializing           y=y.add(control, initialize)
if  Phase=starting               y=y.add(control, collect)
if  Phase=starting
        for each msg_i in invokeMsgs
                                 y=y.add(invoke, msg_i)
if  Phase=collecting  and  repetition≥1
        for each msg_i in invokeMsgs
                                 y=y.add(invoke, msg_i)
if  Phase=stopping               y=y.add(control, passivate)
if  Phase=reporting              y=y.add(control, report)
```

$ta(s) = \sigma$

C.9 Experimental Frame: LCN and DCO Control Instrumentation

This "control instrumentation" specification complements the DEVS specification for the Experimental Frame Acceptor. This behavior is incorporated into the LCN and DCO atomic models to provide experimental frame control instrumentation.

$DEVS_{LCN\_and\_DCO\_Control}$ = < X, Y, S, δ_{int}, δ_{ext}, δ_{conf}, λ, ta >, where

InPorts = { control }

X = { (control, controlMsg) }, where controlMsg = {passivate, initialize}

S = Phase $\times$ σ

Phase = defined in LCN or DCO model

$\sigma = \Re_0^+$

δ_{ext}(s,*e*,(InPorts,X)) = (ph,sigma,) for each x event on "control"

where ph is new Phase, and

sigma is new σ,

if x=passivate ph=passive

sigma=∞

else if x=initialize initialize_LCN_or_DCO_model

δ_{int}(s) = (s)

δ_{conf}(s,ta(s),x) = δ_{ext}(δ_{int}(s),0,x)

λ (s) = $\varnothing$

ta(s) = σ

C.10 Experimental Frame: transducer

$DEVS_{transducer}$= < X, Y, S, δ_{int}, δ_{ext}, δ_{conf}, λ, ta >, where

InPorts = { control }

X = { (control, controlMsg) }

where controlMsg = { initialize, collect, report, passivate}

Y = { (results, report) }

where report is an accounting of the collected data

S = Phase $\times$ σ $\times$ ObservationTime

Phase = { passive, collecting }

$\sigma = \Re_0^+$

ObservationTime = $\Re_0^+$

δ_{ext}(s,*e*,(InPorts,X)) =

(,sigma,OT)	before processing any events
where sigma is new σ,	
sigma = σ-*e*	
if Phase=collecting	*ee*=*e*
else	*ee*=0
(ph,sigma,)	for each x event on "control"
where ph is new Phase, and	
sigma is new σ,	
if x=initialize	initialize_myself
else if x=collect	ph=collecting
	sigma=∞
else if x=report	ph=reporting
	sigma=0
else if x=passivate	ph=passive
	sigma=∞
(,,OT)	after processing all events
where OT is new ObservationTime,	
if Phase=collecting	OT=ObservationTime+*ee*

δ_{int}(s) = (ph,sigma,OT)

where ph is new Phase,	ph=passive
where sigma is new σ,	sigma=∞
where OT is new ObservationTime,	OT=ObservationTime+σ

δ_{conf}(s,ta(s),x) = δ_{ext}(δ_{int}(s),0,x)

λ (s) = y

where y is a set of output (port, value) pairs

y = new empty set

if Phase=reporting	report=generate_my_results_report
	y=y.add(results, report)

ta(s) = σ

APPENDIX D. GLOSSARY OR TERMS

CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
DCO	Distributed Cooperative Object
DEM	Distributed Exercise Manager
DESS	Differential Equation System Specification
DEVS	Discrete Event System Specification
DiSect	Distributed Simulation Exercise Construction Toolset
DMSO	Defense Modeling and Simulation Office
DNS	Domain Name Service
DOC	Distributed Object Computing
DTSS	Discrete Time System Specification
EF	Experimental Frame
ExGen	Exercise Generation
FDDI	Fiber Distributed Data Interface
FIFO	First-In, First-Out
FSM	Finite State Machine
GUI	Graphical User Interface
HCCL	Heterogeneous Container Class Library
HLA	High Level Architecture
HW	Hardware
IETF	Internet Engineering Task Force
ISO	International Standards Organization
JVM	Java Virtual Machine
LAN	Local Area Network
LCN	Loosely Coupled Network
M&S	Modeling and Simulation
MAARS	Modular After Action Review System

MAU	Media Access Units
MIB	Management Information Bases
NIC	Network Interface Cards
OS	Operating System
OSM	Object System Mapping
QoS	Quality of Service
RTI	Run Time Infrastructure
RUP	Rational Unified Process
SES	System Entity Structure
SMTP	Simple Mail Transfer Protocol
SNMP	Simple Network Management Protocol
SW	Software
SwObject	Software Object
UML	Unified Modeling Language
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

REFERENCES

AIS99 Artificial Intelligence & Simulation Research Group (AIS). *DEVSJAVA*, Electrical & Computer Engineering Department, University of Arizona, Tucson, AZ. Available at http://www-ais.ece.arizona.edu/SOFTWARE/, (last visited Sep 1999).

All97 Allen, Robert J. *A Formal Approach to Software Architecture*, School of Computer Science, Carnegie Mellon University, CMU-CS-97-144, May 1997.

But94 Butler, James M. *Quantum Modeling of Distributed Object Computing*, Simulation Digest, 1995. **24**(2): 20-39.

CAC99 CACI. *Comnet III™*, CACI, Arlington, VA. 1999. Available at http://www.caci.com (last visited Jan 2000).

Car99 Carrier, L. *Managing at Light Speed*, IEEE Computer, 1999. **32**(7): 107-109.

Cel91 Cellier, F. E. *Continuous System Modeling*. 1991, New York: Springer-Verlag.

Cha79 Chandy, K.M. and J. Misra. *Distributed Simulation: A Case Study in Design and Verification of Distributed Programs*, IEEE Transactions on Software Engineering, 1979. **5**(5): 440-452.

Chi94 Chiodo, M., et al. *Hardware-Software Codesign of Embedded Systems.* IEEE Micro, 1994. **14**(4): 26-36.

Cho99 Cho, H. *Discrete Event System Homomorphisms: Design & Implementation of Quantized-based Distributed Simulation Environment*, Electrical & Computer Engineering Department, University of Arizona, Tucson, AZ. 1999.

Cho96 Chow, A. *Parallel DEVS: A Parallel, Hierarchical, Modular Modeling Formalism and Its Distributed Simulator.* Transactions on Simulation, Society for Computer Simulation, 1996. **13**(2): 55-102.

Dah98 Dahmann, J.S., F. Kuhl, et. al. *Standards for Simulation: As Simple As Possible But Not Simpler — The High Level Architecture For Simulation*, Simulation, Society for Computer Simulation. **71**(6): 378-387.

DMS98a Defense Modeling and Simulation Office (DMSO). *High Level Architecture Object Model Template (Version 1.3)*, DMSO, U.S. Department of Defense, 1998. Available at http://www.dmso.mil/ (last visited Oct 1999).

DMS98b Defense Modeling and Simulation Office (DMSO). *High Level Architecture Rules (Version 1.3)*, DMSO, U.S. Department of Defense, 1998. Available at http://www.dmso.mil/ (last visited Oct 1999).

DMS99 Defense Modeling and Simulation Office (DMSO). *HLA Homepage*, DMSO, U.S. Department of Defense, 1999. Available at http://www.dmso.mil/hla/ (last visited Oct 1999).

EDL99 Engineering Design Laboratory (EDL) Group. *Hardware/Software Codesign*, Electrical & Computer Engineering Department, University of Arizona, Tucson, AZ. 1999.

Fli99 Fleischmann, J. and K. Buchenrieder. *Prototyping Networked Embedded Systems.* IEEE Computer, 1999. **32**(2): 116-119.

Fox96 Fox, G.C. and W. Furmanski. *Java for Parallel Computing and as a General Language for Scientific and Engineering Simulation and Modeling*. Northeast Parallel Architectures Center (NPAC), Syracuse University, Syracuse NY. 1996. Available at http://www.npac.syr.edu/projects/javaforcse/javameettalks.html (last visited Jan 2000).

GMD99 GMD. *CASTEL*, Available at http://borneo.gmd.de/EDS/SYDIS/castle/start.html (last visited Sep 1999).

Hil98a Hild, D., Sarjoughian, H.S. *Ethernet: DEVS-Based Ethernet Modeling & Simulation*, SCS Western Multi-Conference, Society for Computer Simulation, San Diego, CA. 1998.

Hil98b Hild, D., and H. S. Sarjoughian. *Distributed Cooperative Objects: DEVS-Based Quantum Modeling and Simulation*, SCS Western Multi-Conference, Society for Computer Simulation, San Diego, CA. 1998.

Hil99 Hild, D., H.S. Sarjoughian, and B.P. Zeigler. *Distributed Object Computing: DEVS-Based Modeling and Simulation*. 13th SPIE, Orlando, FL. 1999.

IET90 Internet Engineering Task Force (IETF). *Simple Network Management Protocol (SNMP)*, Standard 15 / Request For Comment (RFC) - 1157, May 1990.

Ins98 Instantiations, Inc. *JOVE: Super Optimizing Deployment Environment™ for Java™*, Technical Report, Instantiations, Inc. July 1998. Available at http://www.instantiations.com/ (last visited Jan 2000).

MIL99 MIL3, Inc. *OPNET™ Modeler*, MIL3, Inc., Washington, DC. 1999. Available at http://www.mil3.com (last visited Jan 2000).

Nun95 Nunamaker, J.F., Jr., R.O. Briggs, and D. D. Mittleman. *Groupware user experience: ten years of lessons with GroupSystems*, in *Groupware: Technology and Applications*, D. Coleman and R. Khanna, Editors. 1995, Prentice Hall: Englewood Cliffs, NJ.

OMG99 Object Management Group (OMG). *What Is OMG-UML and Why Is It Important?* Available at http://www.omg.org/news/pr97/umlprimer.html, (last visited Sep 1999).

Rat99 Rational Software Corporation. *Rational Unified Process: Best Practices for Software Development Teams*, available at http://www.rational.com/ (last visited Sep 1999).

Rou97 Roulo, M. *Java's three types of portability*, Java World, May 1997. Available at http://www.javaworld.com/javaworld/jw-05-1997/jw-05-portability.html (last visited Aug 1999).

Roz94 Rozenblit, J. and K. Buchenrider, eds. *Computer-Aided Software/Hardware Engineering*. 1994, IEEE Press: New York.

Roz99 Rozenblit, J.W., and et. al. *SONORA*, Electrical & Computer Engineering Department, University of Arizona, Tucson, AZ. 1999. Available at http://www.ece.arizona.edu/~edl/sonora.html (last visited Jan 2000).

Sar97 Sarjoughian, H.S. and B.P. Zeigler. *DEVSJAVA: Basis for a DEVS-based Collaborative M&S Environment*. SCS Western Multi-Conference, Society for Computer Simulation, San Diego, CA. 1997.

Sar99a Sarjoughian, H.S. and J. Nutaro, et. al. *Collaborative DEVS Modeler: An Environment Supporting Geographically Dispersed Modelers*, IEEE Systems, Man, and Cybernetics, 1999. (submitted).

Sar99b Sarjoughian, H.S., B.P. Zeigler, et. al. *Collaborative Distributed Network System: A Lightweight Middleware Supporting Collaborative DEVS Modeling*, Future Generation Computer Systems, 1999. (accepted).

Sch97 Schmidt, D. *Guest Editorial: Distributed Object Computing*. IEEE Communications Magazine, Feb 1997. **35**(2): 42-44.

Sch98 Schulz, S., et al. *Model-based Codesign.* IEEE Computer, 1998. **31**(8): 60-67.

STR99 STRICOM, U.S.A. *DiSect: Distributed Simulation Exercise Construction Toolset*, U.S. Department of Defense, 1999. Available at http://www.stricom.army.mil/STRICOM/E-DIR/ES/DISECT/ (last visited Sep 1999).

Wir99 Wirfs-Brock, A. *Complex Java Applications: Breaking the Speed Limit*, Java Report. 1999. **4**: 23-30.

Yen97 Yen, T.-Y., W.H. Wolf. *Hardware-software Co-Synthesis of Distributed Embedded Systems*. 1997, Boston, MA: Kluwer Academic. 156.

Zei84 Zeigler, B. P. *Multi-Facetted Modeling and Discrete Event Simulation*. 1984, New York: Academic Press.

Zei90 Zeigler, B. P. *Object-Oriented Simulation with Hierarchical, Modular Models*. 1990, San Diego: Academic Press.

Zei91 Zeigler, B. P. *Object-Oriented Modelling and Discrete Event Simulation*, Advances in Computers, Ed: M. Yovitz, Academic Press, Boston, 1991. Vol. 33, pp. 68-114.

Zei97a Zeigler, B.P. *Objects and Systems: Principled Design with C++/Java Implementation*. Undergraduate Texts in Computer Science, ed. D. Gries. 1997, New York, NY: Springer-Verlag.

Zei97b Zeigler, B.P., et.al. *The DEVS Environment for High-Performance Modeling and Simulation*, IEEE Computer Science and Engineering, 1997. **4**(3): 61-71.

Zei98 Zeigler, B.P. and J.S. Lee. *Theory of Quantized Systems: Formal Basis for DEVS/HLA Distributed Simulation Environment*. in *Enabling Technology for Simulation Science(II)*, SPIE AeoroSense 98. Orlando, FL. 1998.

Zei99a Zeigler, B.P., et al. *Bandwidth Utilization/Fidelity Tradeoffs in Predictive Filtering*, Simulation Interoperability Workshop, March 1999. Orlando,FL.

Zei99b Zeigler, B.P., et al. *Implementation of the DEVS Formalism over the HLA/RTI: Problems and Solutions*, Simulation Interoperability Workshop, March 1999. Orlando,FL.

Zei99c Zeigler, B. P., et al. *Predictive Contract Methodology and Federation Performance*, Simulation Interoperability Workshop, September 1999, Orlando, FL.

Zei00 Zeigler, B. P., Kim, T. G., and Praehofer, H. *Theory of Modelling and Simulation*, Academic Press, 2nd Edition, 2000.