The Pennsylvania State University

The Graduate School

Department of Industrial and Manufacturing Engineering

AN ADAPTIVE FRAMEWORK

FOR

SYNCHRONIZATION OF DISTRIBUTED SIMULATIONS

A Thesis in

Industrial Engineering

by

Bertan Altuntas

© 2005 Bertan Altuntas

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

May 2005
The thesis of Bertan Altuntas was reviewed and approved* by the following:

Richard A. Wysk  
Professor and Leonhard Chair in Engineering  
Thesis Advisor  
Chair of Committee

Catherine M. Harmonosky  
Associate Professor of Industrial and Manufacturing Engineering

Vittaldas Prabhu  
Associate Professor of Industrial and Manufacturing Engineering

David L. Hall  
Associate Dean for Research, School of IST

Richard J. Koubek  
Professor of Industrial and Manufacturing Engineering  
Head of the Department of Industrial and Manufacturing Engineering

*Signatures are on file in the Graduate School
ABSTRACT

Increased complexity of simulation models has created the need for distributing the execution of simulations on multiple processors. While distributed simulation promises rapid sub-model development and faster execution speeds, it poses two challenging problems: (a) synchronization of distributed simulations and (b) rapid development and deployment of integrated distributed simulation systems.

In this doctoral research, a new adaptive synchronization framework was developed to address both of the aforementioned problems. Formal models of distributed discrete-event simulation were created based on the finite-state machine representations of discrete-event systems. These models were used for: (a) analyzing the requirements of distributed simulation synchronization and (b) creating an automated synthesis mechanism for rapid development and deployment of distributed simulation systems.

The developed synchronization framework features a novel synchronization protocol, which is based on adaptive pacing of the execution of distributed simulations during interactions. This new synchronization protocol, called Adaptive Partial Pacing (APP) synchronization, relies on synchronized real-time clocks at distributed simulations for coordinated pacing and also features an adaptive pace determination scheme for greater synchronization efficiency.

The developed synchronization framework also features an automated synthesis mechanism which allows rapid creation of a synchronization system from simulation models. This capability allows users with limited specific knowledge of distributed simulation synchronization to create and run distribute simulation systems in a short amount of time. The synchronization framework was successfully implemented using the Arena™ simulation package from Rockwell Automation. The research shows that for loosely coupled simulations, a
significant speed-up of the system can be realized. It also shows that for systems with little variability more significant improvements can be achieved.
# TABLE OF CONTENTS

LIST OF FIGURES .................................................................................................................. viii

LIST OF TABLES .................................................................................................................... xi

ACKNOWLEDGEMENTS .......................................................................................................... xiii

Chapter 1  Introduction ........................................................................................................ 1

1.1 Key Concepts and Definitions ...................................................................................... 4
1.2 Problem Definition and Scope .................................................................................... 7
1.3 Research Objectives and Overview of the Methodology ......................................... 10
1.4 Research Hypothesis and Motivation ....................................................................... 13

Chapter 2  Background .................................................................................................... 22

2.1 Brief History of Parallel and Distributed Simulation ............................................. 22
2.2 Conservative Synchronization Mechanisms ......................................................... 25
2.3 Optimistic Synchronization Mechanisms ............................................................. 28
2.4 Adaptive Synchronization Mechanisms ................................................................ 31

Chapter 3  Structural Models and Analyses of Synchronization Requirements .......... 35

3.1 Basic Structural Model ............................................................................................. 35
3.2 DSS Structural Model ............................................................................................. 45
3.3 Timing of Events ....................................................................................................... 50
3.4 Event Dynamics and Causal Precedence ............................................................... 52
3.5 Scalability and Modeling Power of Structural Models .......................................... 54
3.6 The Property of Synchronization in Distributed Simulation Systems .................. 54
3.7 Analyses of Synchronization Requirements for Paced Execution ....................... 57
   3.7.1 Ideal Case – Perfect Clock Synchronization ................................................ 59
   3.7.2 Realistic Case – Imperfect Clock Synchronization ........................................ 62

Chapter 4  Automated Synthesis of Structural Models ..................................................... 69

4.1 Implementation Domain and Modeling Assumptions ........................................... 71
4.2 A Sample Structural Model ..................................................................................... 72
4.3 Synchronization Architecture and Structural Models ............................................. 81

Chapter 5  Synchronization Implementation Framework ............................................... 86

5.1 Synchronization System Architecture ..................................................................... 86
5.2 Components of a Local Synchronization Agent .................................................... 94
   5.2.1 The Interaction Management Engine (IME) .................................................. 95
   5.2.2 The Simulation Control Engine (SCE) ........................................................ 98
C.6 EventSet Class ................................................................................................212
C.7 InteractionMessage Class ...............................................................................212
C.8 Message Class ................................................................................................213
C.9 State Class ......................................................................................................213
C.10 StateTimeTable Class ..................................................................................214
C.11 StateType Enumeration................................................................................214
C.12 StateSet Class...............................................................................................215
C.13 TransitionMap Class ....................................................................................215

Appendix D  Glossary .............................................................................................216
LIST OF FIGURES

Figure 1-1: An example system: two-server queuing network ........................................ 14
Figure 1-2: Time advance diagram for LP₁ and LP₂ ......................................................... 17
Figure 3-1: A simple 2-server queuing system ............................................................... 36
Figure 3-2: Connectivity graph for Example 3.1 ............................................................ 37
Figure 3-3: Directed graph representation of the structural model ................................ 40
Figure 3-4: Directed graph representation of $S_{ext}$ ..................................................... 41
Figure 3-5: Combination of two entity specific models ................................................. 46
Figure 3-6: Advance of LVC in event-driven simulation execution .............................. 52
Figure 3-7: Worst case behavior of system clocks during execution ............................... 64
Figure 4-1: Graphical representation of the example physical process (PP₁) .......... 73
Figure 4-2: Arena™ simulation model – graphical representation ............................... 75
Figure 4-3: Graphical representation of the general structural model ....................... 77
Figure 4-4: Graphical representation of the $\varepsilon_L$-specific structural model .......... 79
Figure 4-5: Graphical representation of the $\varepsilon_g$-specific structural model ........... 80
Figure 4-6: LSA structure and DSS architecture ............................................................. 83
Figure 4-7: Synthesis of structural models ..................................................................... 84
Figure 5-1: General Architecture of the Synchronization Framework ....................... 92
Figure 5-2: Anatomy of a local synchronization agent ................................................. 94
Figure 5-3: Lifecycle of a single isolated interaction ...................................................... 106
Figure 5-4: Shape of the $f$ transformation ................................................................. 112
Figure 5-5: Shape of the $g$ transformation ................................................................. 113
Figure 5-6: Cross-component relational categorization of auxiliary services and functions ................................................................. 120
Figure B-1: Graphical Representations of Structural Model Components................207
# LIST OF TABLES

Table 1-1: Execution trace of the example DSS ................................................................. 15
Table 1-2: Modified execution trace of the example system’s simulation .................. 16
Table 2-1: Major Synchronization Protocols ................................................................. 34
Table 4-1: Processing time of the parts in the example system (minutes) ................. 74
Table 4-2: Formal definition of the general structural model ..................................... 76
Table 5-1: Message classes and member objects ............................................................ 88
Table 5-2: Definition of the “i-message” class .............................................................. 89
Table 5-3: Definition of the “s-message” class ............................................................. 90
Table 5-4: Definition of the “notice” class ................................................................. 91
Table 5-5: Pseudo-code notation and generic names .................................................. 117
Table 5-6: Flowchart legend ....................................................................................... 118
Table 5-7: The EndSync algorithm - code ................................................................. 121
Table 5-8: The ReceivePIM algorithm - code ............................................................ 123
Table 5-9: Send and Broadcast algorithms – code .................................................... 125
Table 5-10: ClockSync algorithm – code ................................................................. 126
Table 5-11: Start Timer and Stop Timer algorithms – code .................................... 128
Table 5-12: UpdateSMC algorithm – code ............................................................... 130
Table 5-13: Start Pacing and Stop Pacing algorithms – code ................................ 132
Table 5-14: EndSync algorithm - code ................................................................. 133
Table 5-15: UpdatePIM algorithm – code ............................................................. 135
Table 5-16: UpdateCAM algorithm – code .............................................................. 137
Table 5-17: UpdateAPI algorithm – code .............................................................. 139
Table 5-18: Start Sync algorithm – code ............................................................... 141
Table 5-19: SendPOM algorithm – code................................................................. 144
Table 5-20: MAIN algorithm - code ........................................................................ 146
Table 5-21: Initialize algorithm - code ............................................................. 149
Table 5-22: RegularExecute algorithm - code................................................. 152
Table 5-23: PacedExecute algorithm - code.................................................... 154
Table 5-24: Terminate algorithm - code............................................................... 155
Table 6-1: Specifications of the Distributed Simulation System Components........ 163
Table 6-2: Values of the parameters used in time-scaling factor calculations ....... 164
Table 6-3: Experimental design for variability of state-delay analysis (LP₁)........ 168
Table 6-4: Results of the experiments for variability of state-delay analysis......... 168
Table 6-5: Experimental design for factor 1 (LP₁)................................................ 172
Table 6-6: Results of the experiment for degree-of-coupling analysis – factor 1 ...... 172
Table 6-7: Experimental design for factor 2.......................................................... 173
Table 6-8: Results of the degree-of-coupling experiments - factor 2 ................. 174
Table A-1: Important element modules of Arena™ simulation package .......... 198
Table A-2: Important modules of the Arena™ simulation package.................. 200
Table B-1: Mapping of Arena™ modules to Structural Model Components........ 206
ACKNOWLEDGEMENTS

First and foremost, I would like to thank Dr. Richard Wysk for his support, patience and encouragement throughout the course of my graduate studies at the Pennsylvania State University. His ideas and positivism made this work possible and I am honored to be a student of such a good professor and an example person.

I would like to thank my committee members Dr. Catherine Harmonosky, Dr. Vittaldas Prabhu and Dr. David Hall for their help in shaping the research and finalizing this dissertation. Their principles, helpful suggestions and extensive experience made this dissertation complete.

I would like to thank my dear friends Melike Öz and Uğur Paşaoğulları for their sincerity, help and hospitality; and my comrade Seung Yub Lee for sharing the joys and sorrows of research with me.

Last but not least, my wife Ece Erdoğmuş and my mother İnci Tekeli deserve very special thanks for their support and endless love. I dedicate this work to these special women who gave meaning to my life.
Chapter 1

Introduction

One of the fundamental goals in computer technology is to increase the speed of computers. Speed of computation is of utmost importance since almost all real-life scientific and engineering problems are very complex and they demand intensive calculations.

As the power of computers improve, it becomes possible for scientists and businesses to create solutions for very complicated problems in shorter times. However, the current semi-conductor technology limits the economic manufacture of very fast microprocessors. High-end parallel microprocessors are custom manufactured and their costs can be as high as millions of dollars. Consequently, as opposed to these expensive high-end systems, distributed parallel processing becomes a very viable way of creating high performance computation systems.

Current computer networking technology makes it practical to create clusters of concurrently running microprocessors providing much higher computational power (floating point operations per unit time) than single processors [Garg, 1996]. Many businesses and universities around the globe have built their own super computers by creating clusters of many processors connected via high-speed communication networks. For example, The Pennsylvania State University has a computer cluster named Lion-XL composed of 128 Pentium 2.4 GHz processors providing a very powerful computational resource for researchers [LIONXL].

Cost-effectiveness and availability of concurrent computer systems have created widespread application areas for parallel and distributed processing. Computer simulation is one of the areas where parallel and distributed processing has already proved to be an
economical way of providing the performance and storage required for shortening the run times and increasing the fidelity of simulation models. Industries utilizing extensive computer simulations have already implemented many real-world distributed simulation systems for simulating complicated systems.

For example, a prominent automobile manufacturer, BMW, has been using a 700-processor computer cluster to run its crashworthiness simulations [BMW]. BMW’s computer cluster and Penn State’s LION-XL are both examples for tightly coupled parallel computing systems where processors are connected via a super-fast network and confined to a single room or building.

However, concurrent computer systems can also be built using processors, which are geographically distributed that communicate via the Internet. This kind of a distributed system is called loosely coupled since the processors forming the system do not share a common memory and/or storage medium. There are various applications of geographically distributed simulation such as military battlefield simulations, industrial supply chain logistics simulations, air traffic simulations and simulations of telecommunications networks. An extensive review of the application areas is given in [Fujimoto, 2000].

Distributed simulation is an especially good fit when a distributed system such as a supply chain is to be simulated as a whole. Supply chain simulation is a valuable decision support tool, and it is widely used in industry. When a supply chain is simulated, all of the companies involved in the supply chain must provide details of their manufacturing and distribution systems. In the case where a company must protect proprietary information regarding details of its operations (such as cost information), simulation of its operations is limited and it becomes impossible to create a realistic and high fidelity simulation of the whole supply chain.
Using distributed simulation, independent entities involved in the supply chain do not need to merge their simulation models into one big model. They only need to create interfaces so that their models can interact with the other models. Therefore, details of their operations become transparent to the other entities in the broader system. This also makes it possible for the companies to use existing simulation models without significant modification, except for the inclusion of necessary interfaces for communication.

In some cases, even a single system can be too complicated to capture into a single simulation model. High fidelity models of manufacturing systems can become intractable and run slower than the real system. A well known method to deal with complexity is to divide the system into smaller manageable pieces and model separately—model decomposition. As long as these smaller pieces can interact with each other, distributed simulation can be used to combine these smaller models to create a whole model of the system.

Above mentioned issues and continuously increasing performance demands of the general scientific community make distributed simulation an increasingly important tool and therefore research in this area has gained considerable attention in the last few decades. Several different architectures and protocols for implementing parallel and distributed simulation systems have been developed over the years.

This research focuses on one of the most challenging problems of parallel and distributed simulation: synchronization or also known as time management.

**Definition 1.1:** Synchronization – in the context of distributed discrete event simulation – is the act of coordinating dependent simulation processes running on different computers so that causal ordering of all events is protected. It is also called time management, because the coordination requires control of individual simulation clocks.
Synchronization is of utmost importance in the design of a distributed simulation system, since the accuracy of the results generated by the distributed simulation can depend on the reliability of the synchronization mechanism used. Lack of synchronization can cause the incorrect ordering of events, which in turn will cause inaccurate results.

Unlike a simulation that runs on a single computer, the need for synchronization in a distributed computing environment creates increased overhead because of the extra processing and communication required for coordination. Therefore, efficiency and speed of a distributed simulation is highly dependent on the type and quality of the synchronization mechanism used.

The remainder of this chapter is organized as follows. Section 1.1 introduces the main concepts in distributed simulation and defines the major keywords used throughout this dissertation. In Section 1.2, the research problem, namely the problem of synchronizing distributed discrete event simulations, is defined. Section 1.3 lists specific objectives of this dissertation and gives a brief overview of the methodology used to achieve the objectives. Finally, the research hypothesis and motivation for conducting this research is provided in the last section.

1.1 Key Concepts and Definitions

In order to provide a consistent discussion, a set of keywords has been defined to identify the key concepts and elements of this research. This section provides definitions of the keywords and is intended to serve as a guide to the terminology used in this dissertation. Also, for quick reference, a glossary of all important keywords is provided in Appendix D.

In several places throughout this dissertation, acronyms are used instead of full keywords to provide succinct arguments and avoid excessive repetitions of certain words.
These acronyms are also defined in this section and listed in Appendix D. There are other acronyms and keywords not included here which are defined in their relevant chapters. Definitions of these can be found in Appendix D.

Some keywords and terms are borrowed from [Fujimoto, 2000] to be consistent with the existing parallel and distributed simulation literature.

A **distributed simulation system (DSS)** is a collection of at least two interacting simulation processes which run on different computers that are connected via a communication network along with necessary services to manage the simulation processes. In order to distinguish a distributed simulation from a simulation that runs on a single computer, the latter is called a **sequential simulation**, since execution is not done concurrently on several computers but sequentially on a single computer.

A **simulation process** or a **logical process (LP)** in a distributed simulation system is a computer program which implements a simulation model. Simulation processes in a DSS have the capability of sending and receiving information using the communication network of the DSS.

A **simulation model** is an abstraction of a real or an imaginary system, which is developed usually with the intention of analyzing the system’s behavior. The system being modeled is called a **physical process (PP)**.

The formalism used to create simulation models is called a **simulation language**. Although, the term simulation language, in its most common meaning, represents languages embedded in commercial simulation packages (such as SIMAN of Arena™), any generic modeling formalism (such as Petri-Nets and Markov Chains) can be regarded as a simulation language as long as there are instruments/methods available to create simulation processes from its instances.
In summary, each physical process is represented by a logical process and the interactions among the physical processes are represented by exchanging messages between logical processes.

Physical processes interact by exchanging **physical entities**. For example, in a supply chain, goods that are shipped between suppliers and retailers are physical entities and their shipment affect the state of the retailers (inventory levels increase). Physical entities need not always be tangible items. For example, in a battlefield, orders issued to battalions from the headquarters by means of radio communication are also termed physical entities.

In general, any tangible or intangible item that has an effect on the state of the local or remote physical processes can be considered as a physical entity. Physical entities are represented by **logical entities** or **simulation entities** in the logical processes. Therefore, interactions in the DSS imply exchange of simulation entities between logical processes.

An interaction between the physical processes requires a certain amount of time (speed of any interaction in real-life is limited by the speed of light). This delay between the initiation and completion of an interaction (e.g. shipment of an order and its arrival at the retailer) is called the **physical interaction delay (PID)**. In the DSS, this is represented by a simulated delay called the **logical interaction delay (LID)**. Interaction delays play an important role in the synchronization of distributed simulations.

Each logical process has a simulation clock, which is called the **Local Virtual Clock (LVC)**. In order to distinguish simulation clocks from the real clock, the real-time clock will be referred to as the **wallclock**. The value of a local virtual clock at an instance of the wallclock is called the **Local Virtual Time (LVT)**, while the value of the wallclock is simply called the **time**.
An execution of a logical process is an orderly processing of events that represent actions in physical processes. Every event is attached a timestamp that indicates the scheduled local virtual time of execution for that event. An execution trace is an ordered list of events occurred during an execution of an LP.

At the beginning of an execution, a set of initial events are created. After the execution starts, processing of events cause creation and scheduling of new events. Events created locally in this manner are called local events since they originate at the local logical process.

Events can also be created and scheduled for execution by means of receiving messages from remote LPs. This is how interactions take place in the DSS. Events that are created by received messages are called remote events since they originate at a remote logical process.

A message is a compact piece of information that represents the specifics of a particular interaction between two logical processes. The type of information encapsulated in a message depends on the specifics of the DSS environment. Usually, data regarding the entity being transferred and timing of the local and remote events related to the message forms the essential information contained in a message.

### 1.2 Problem Definition and Scope

Logical processes in a distributed simulation system interact by sending/receiving messages. Under normal conditions, messages require a finite amount of time in the communication network to travel to their destinations.

In general, the amount of time a message takes to travel from one LP to another is not constant and depends on many parameters such as the distance between the computers hosting LPs, network load and network behavior at the time of transmission.
Therefore, two messages sent at the same time from two different LPs to another common LP may arrive at the destination LP in any order due to the non-deterministic nature of speed of communication.

An LP receiving messages can process a later sent message before it even receives an earlier sent message. This situation is known as a time ambiguity, since a possible causal relationship between these two messages would be violated due to the incorrect order of event execution at the target LP.

In fact, even if one can guarantee proper ordering of messages during transmission (assuming that there is a dedicated communication network that assures constant transmission times between any two LPs for any message), it is still possible to have time ambiguities. Because the sufficient condition for a time ambiguity to occur is execution of causally related events in incorrect order.

In a sequential simulation, causality between events is always protected because there exists a data structure (usually called an event list or calendar) that stores all current and future scheduled events and executes them in increasing timestamp order. However, in a true distributed simulation system there does not exist a global data structure that holds all events (otherwise it becomes a sequential simulation that is distributed over multiple computers yet executed sequentially).

Consequently, although each LP manages its local events using its own event ordering mechanism, it does not have any control over remote events. Because, remote events are externally inserted into the local event list of an LP, which immediately implies that the timestamp of a remote event is not necessarily greater than the timestamp of the first event in the local event list.

If this happens, the remote event becomes the first event to be executed by the logical process. However, if earlier (before the receipt of this remote event) an event with
a timestamp greater than the timestamp of this remote event was executed, this means that the local virtual time of the LP is past the scheduled execution time of the remote event. Obviously, execution of the remote event creates a time ambiguity, because it violates the causal ordering of events.

The root cause of this problem lies in the execution speeds of LPs in the DSS. More specifically, if local virtual clocks of LPs advance independently at different speeds, eventually, a gap between the clocks of fast and slow LPs will form. When this gap between LVCs grows greater than the maximum logical interaction delay between these two LPs, any remote event originating at the slow LP and directed to the fast LP will create a time ambiguity. These remote events are called straggler events.

Straggler events and time ambiguities caused by them should either be completely avoided throughout the entire simulation run or upon occurrence they should be resolved in run-time.

In the light of these observations, it is clear that there is a need for managing the execution of individual logical processes as well as the way messages are sent, received and processed. Namely, the interactions among logical processes should be synchronized.

This research is focused on the synchronization of distributed discrete-event simulations with applications in modeling and analysis of complex systems (i.e., systems of systems).

Two main approaches have been used in the distributed simulation synchronization: conservative synchronization and optimistic synchronization. Conservative synchronization methods guarantee continuous synchronization by controlling the execution of unsafe events (events that may cause a time ambiguity) during run-time.
Conservative synchronization methods sacrifice from concurrency to assure continuous synchronization.

On the other hand, optimistic synchronization methods do not restrict the execution of individual logical processes. Instead, these methods allow the logical processes to run as fast as possible and intervene when a time ambiguity occurs in the system. Upon occurrence of the ambiguity, the faulty logical process is stopped and a rollback is executed on all effected LPs to recover from the error. Once the system recovers from the time ambiguity, execution resumes.

Optimistic methods are known to perform better than conservative methods when the logical processes are loosely coupled and interactions between them are infrequent. A more detailed review of existing synchronization techniques is provided in Chapter 2.

1.3 Research Objectives and Overview of the Methodology

The main objectives of this research can be summarized as follows:

1. To develop a formal analysis of the synchronization property as well as the requirements for time-stepped synchronization by using mathematical models of distributed discrete-event systems.
2. To create a finite state automata based modeling methodology and demonstrate a formal method of automated synthesis and integration of synchronization agents for simulation models. Synchronization agents control the speed of execution of LPs to provide continuous conservative synchronization.
3. To create and demonstrate a conservative and adaptive synchronization algorithm and an architecture for its application based on the requirements obtained from the formal analysis of time-stepped synchronization.
This research does not seek to create the most efficient synchronization mechanism for distributed discrete-event simulations. The focus of this dissertation is on developing a scalable and yet reasonably efficient synchronization framework which can be used to develop distributed simulation systems with minimum ease. Although efficiency is a central problem to the research of parallel computing, there are many other benefits of easy and rapid parallelization such as ease of high-fidelity model development and distributed development of complex simulation models.

Existing formal models of distributed systems and discrete-event systems are considered to develop the mathematical tools to be used in the analysis of the synchronization property. Automata theory is determined to be a suitable tool for modeling both distributed and discrete-event aspects of DSSs.

A modified version of finite state automata (timed communicating state automata) is developed to model the dynamics of logical processes regarding the synchronization of a distributed simulation system. This formal model of a logical process incorporates a timing structure, a messaging structure and other auxiliary structures to define and distinguish between different events and states of logical processes.

A composition operation is defined in the set of models of LPs which is later used to create the formal model of a distributed simulation system. This automata based formal model of a DSS is used to analyze the synchronization property and determine the requirements for creating a time-stepped synchronization mechanism.

An adaptive time-stepped (paced) synchronization algorithm is created based on the findings from the formal analysis and using some additional analysis regarding the synchronization of hardware clocks and error compensation. The algorithm is designed to run in a distributed architecture.
In this architecture, each LP is coupled with a “Local Synchronization Agent (LSA)”. LSAs provide necessary functions for detection of imminent interactions (and prediction of interactions times), communication services and the control of the execution speed of LPs.

The synchronization algorithm runs at the heart of every LSA. LSAs predict imminent interactions originating at their coupled LPs and periodically communicate this information to other LSAs using special messages (not to be confused with the messages used for interaction between LPs). Using this distributed awareness mechanism, a list of all imminent interactions in the DSS is collectively established at each LSA.

The synchronization algorithm reads the list of imminent interactions and calculates the safe speed of execution for the upcoming interaction. Since the same synchronization algorithm is run at each LSA and all LSAs have the same interaction list, an automated consensus for the speed of execution is reached among all LSAs before the interaction starts.

At the predicted start time of the current interaction, each LSA reduces the speed of its coupled LP to the pre-calculated execution speed. Once the interaction is complete, LSAs communicate to synchronously switch to regular event-driven execution until the next interaction.

This synchronous change of execution speed in the system combined with a mechanism to synchronize hardware clocks forms the overall adaptive paced synchronization framework.

The most important and model dependent part of an LSA is the data structure and algorithm that monitors the state of the coupled logical process and predict the imminent interactions originating from the LP. Based on the modified automata models defined for the analysis of the synchronization property, an automated synthesis mechanism is
developed. This mechanism maps existing simulation models to an automata based structural model that is directly embedded in the LSA and used for monitoring the state of the simulation process and predicting the timing of upcoming interactions.

All of the created data structures and algorithms were implemented using Visual C++ (v. 6.0) and Visual C# (.NET) compatible with the Windows XP operating system and Arena™ simulation package (v. 7.01) from Rockwell Software. The resulting system is capable of automatically generating local synchronization agents from a set of Arena™ simulation models designed to run in a distributed setting connected to a local area network.

1.4 Research Hypothesis and Motivation

In order to illustrate the challenge of synchronization the following example is provided.

Example 1.1: Consider a simple queuing network composed of two FIFO servers in tandem with infinite queue capacities and external arrival events as seen in Figure 1. Assume that the interarrival times for LP_1 and LP_2 are 4 and 5 minutes and the service times are 2 and 8 minutes, respectively. Entities departing from server 1 (S_1) are transported to server 2 (S_2) and transportation of the entities from S_1 to S_2 takes 1 minute.
Assume that the DSS used to simulate this system has two LPs (LP₁ and LP₂) each representing one server. For simplicity, assume that the speeds of the computer processors running the LPs are identical (i.e. both computers process the same amount of events per second).

In this system, three event types can be defined: **arrival**, **service start** and **departure**. As an entity goes through the simulation, it schedules these three events in order. When an entity is leaving S₁, the departure event executed on LP₁ sends a message to LP₂ in order to schedule a remote arrival event for the leaving entity. This message has a timestamp of the sum of the LVC_{LP₁} at the instant of sending of the message and the transportation time (1 minute). This timestamp tells LP₂ when to schedule the arrival event associated with this new entity coming from LP₁.

Each entity is numbered by a unique integer. Entities originating at LP₁ are numbered starting from 100 and those originating at LP₂ are numbered starting from 200. Table 1 shows the execution trace of this DSS for six steps. Here, each row represents a step of execution and the event being executed is denoted by the 3-tuple <entity number,
event type, event timestamp>. Event types are abbreviated as: arrival – a, service start – ss and departure – d.

Table 1-1: Execution trace of the example DSS

<table>
<thead>
<tr>
<th>Execution Step</th>
<th>LP₁</th>
<th>LP₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>e₁₁ = &lt;101,a,0&gt;</td>
<td>e₂₁ = &lt;201,a,0&gt;</td>
</tr>
<tr>
<td>2</td>
<td>e₁₂ = &lt;101,ss,0&gt;</td>
<td>e₂₂ = &lt;201,ss,0&gt;</td>
</tr>
<tr>
<td>3</td>
<td>e₁₃ = &lt;101,d,2&gt;</td>
<td>e₂₃ = &lt;101,a,3&gt;</td>
</tr>
<tr>
<td>4</td>
<td>e₁₄ = &lt;102,a,4&gt;</td>
<td>e₂₄ = &lt;202,a,5&gt;</td>
</tr>
<tr>
<td>5</td>
<td>e₁₅ = &lt;102,ss,4&gt;</td>
<td>e₂₅ = &lt;201,d,8&gt;</td>
</tr>
<tr>
<td>6</td>
<td>e₁₆ = &lt;102,d,6&gt;</td>
<td>e₂₆ = &lt;102,a,7&gt;</td>
</tr>
</tbody>
</table>

Each departure event at LP₁ sends a message to LP₂ that schedules an arrival event. For simplicity, assume that the communication delay during messaging is negligible. The first message m₁ is sent at time 2 from LP₁ to LP₂ and this message scheduled an arrival event at time 3 in LP₂. This event (e₂₃) is executed right away since it has the earliest timestamp in the event list (the other event in the list at this step is e₂₄, which was scheduled at step 1 by the arrival of the first entity).

Event e₂₃ is executed with no problem because LVC_{LP₂} was still zero when m₁ was received. However, the second message that was sent at time 6 schedules an arrival event at time 7 in LP₂, although LVC_{LP₂} has already advanced to 8 in the previous step by executing e₂₅. Therefore, at this point, a time ambiguity occurs in LP₂. If the execution continues, the simulation results will be biased and will not reflect the real behavior of the system.

This example clearly shows that a time ambiguity occurs because the simulation clocks of LP₁ and LP₂ advanced independently at different and irregular rates. In order to avoid the time ambiguity at time 8, the execution of step 5 at LP₂ can be delayed until
LP₂ receives the message m₂ from LP₁. Table 2 shows the modified execution trace with inserted idle time at step 5 of the execution of LP₂.

The solution presented in Table 2 is one possible way of avoiding the time ambiguity for this instance. However, note that this solution approach requires LP₂ (the receiving LP) to know in advance that LP₁ is going to send a message with a timestamp less than its next event in the list waiting to be executed. In general, it is not a trivial problem to predict when to expect a message from the source LPs (message senders).

Table 1-2: Modified execution trace of the example system’s simulation

<table>
<thead>
<tr>
<th>Execution Step</th>
<th>LP₁</th>
<th>LP₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>θ₁₁ = &lt;101,a,0&gt;</td>
<td>θ₂₁ = &lt;201,a,0&gt;</td>
</tr>
<tr>
<td>2</td>
<td>θ₁₂ = &lt;101,ss,0&gt;</td>
<td>θ₂₂ = &lt;201,ss,0&gt;</td>
</tr>
<tr>
<td>3</td>
<td>θ₁₃ = &lt;101,d,2&gt;</td>
<td>θ₂₃ = &lt;101,a,3&gt;</td>
</tr>
<tr>
<td>4</td>
<td>θ₁₄ = &lt;102,a,4&gt;</td>
<td>θ₂₄ = &lt;202,a,5&gt;</td>
</tr>
<tr>
<td>5</td>
<td>θ₁₅ = &lt;102,ss,4&gt;</td>
<td>idle</td>
</tr>
<tr>
<td>6</td>
<td>θ₁₆ = &lt;102,d,6&gt;</td>
<td>θ₂₅ = &lt;102,a,7&gt;</td>
</tr>
<tr>
<td>7</td>
<td>...</td>
<td>θ₂₆ = &lt;201,d,8&gt;</td>
</tr>
</tbody>
</table>

In general, one can say that the root cause of the time ambiguities is the asynchronous execution of LPs. Asynchronous execution means that the simulation clocks of federates advance independently from each other. Figure 2 shows the course of the LVCs of LP₁ and LP₂ for the queuing network example. Notice how LVC_LP₂ outruns LVC_LP₁ and causes a time ambiguity at time 8. Any message received in the shaded area will cause a time ambiguity at LP₂.

It is clear that if LVC of a receiving LP advances faster than that of the source LP then it is highly likely that a message coming from the source LP will cause a time
ambiguity. Existing conservative synchronization mechanisms avoid this situation by blocking the receiving LP until the gap between the LVCs reduces to a safer value.

A natural solution to this problem is to execute the source and receiving LPs “synchronously”. If a mechanism can advance LVCs in a coordinated fashion (at a common rate) during the execution, it becomes quite easy to avoid time ambiguities.

A straightforward way of advancing all simulation clocks synchronously is to use a time-stepped execution mechanism. In time-stepped execution, at each step all simulation clocks advance by a constant value and at the end of the step all events/messages occurred during that step is executed. Start and end of a step is coordinated by passing messages among the LPs. Although this mechanism is fairly easy to implement, it creates a considerable amount of communication overhead. This is because all LPs must send at least one message for coordination purposes at each step. Considering that communication is the main bottleneck in a distributed computing system, communication complexity of this synchronous time advance mechanism hinders efficient execution of the DSS.
In order to reduce this type of communication overhead, it is possible to use a reference real-time clock to pace the advance of time-steps, which is the technique developed in this research. If all the simulation clocks can be advanced at a common rate synchronized with the rate of advance of a reference wallclock, then it is possible to advance the clocks of the federates at a fairly synchronous way. Such a mechanism can significantly reduce the amount of communication for coordination compared to the above discussed technique, since LPs only communicate to the reference clock when their local hardware clocks drift far apart enough to endanger synchronicity.

Time-stepped distributed simulation has one of its first applications in [Peacock et al., 1979]. Righter and Walrand [1989] provided some insight into the potential benefits of time-stepped distributed simulation. They also gave a clue of adaptive time-driven distributed synchronization without a detailed discussion of the idea. Prabhu and Duffie [1995] developed a distributed control architecture, which uses a replica of the control software as a distributed simulation model. The simulation is executed faster than real-time for analysis, by using time-stepped execution. In this work, a scaling factor is used for speeding up the execution of the distributed simulation system in a consistent manner.

Time-stepped simulation is not the most efficient way of execution when sequential discrete event simulations are considered. However, in a distributed setting it provides a valuable mechanism for synchronous execution. In time-stepped distributed simulation, each computer can execute its logical process synchronized with a global clock. The global clock can run at wallclock speed or at a scaled version of the wallclock speed (slower or faster). Therefore, if each computer can synchronize its internal clock to the global clock, all processors can execute their local simulation processes at the same speed.

This immediately implies that all logical processes should have the same local virtual clock value at any time during the execution (assuming that they synchronously start with the same value) with some tolerance due to processor clock synchronization. In
this case, if the communication delays of messages between the LPs are small compared to the simulated virtual time between the events (the time between two remote events that are causally related) then there is no possibility for a time ambiguity to occur at any one of the LPs.

These observations imply that if proper execution speeds are selected so that the processing times in LPs are comparably greater than the amount of communication latency then time-stepped simulation can be used to create synchronized distributed simulation systems. A few recent studies in the simulation based control area have pointed out the ease of synchronizing distributed discrete event simulations using time-stepped execution [Mahalati, 2002 and Cho, 2000]. However, these studies have all utilized time-stepped simulation in the periphery of a control system application (as an auxiliary analysis tool). Sharma [2001] developed an initial analysis of the limit on the speed-up of a time-driven distributed simulation system.

There is a gap in the distributed simulation literature in terms of exploiting the real potential and analyzing the limitations of time-stepped synchronization. There are several possible approaches to improve the efficiency of time-stepped synchronization, which have not been addressed in the literature yet. One approach is to dynamically adjust the time step size (as stated in [Righter and Walrand, 1989]) or in other words change the speed of execution.

Das [2000] and Srinivasan and Reynolds [1995 and 1998] have discussed the potential superiority of using an adaptive technique in which the parameters for synchronization are adjusted dynamically during the execution of the distributed simulation. Both studies rightfully suggest that stochastic simulations have very dynamic and unpredictable execution paths. Therefore, in many cases, a synchronization approach that uses static information about simulation models would be less efficient compared to a dynamic version of the same approach that evaluates the amount of interactions
between simulation processes, while continually fine-tuning the engagement of the synchronization controls for faster and smoother execution.

In this respect, the speed of execution in a time-stepped distributed simulation can be dynamically adjusted on the fly among the LPs. This idea forms the main starting point and motivation for this research.

This research also exploits another advantage of time-stepped synchronization: automation of synthesis and integration of the synchronization mechanism. In a panel addressing the strategic directions in simulation research, Fujimoto, one of the most prominent researchers in the field, stated that the driving force behind the academic research in distributed simulation has become practical and rapid development of parallel simulation systems [Page et al., 1999]. In another seminal paper, Fujimoto suggested that current distributed simulation mechanisms are not practical to be utilized by non-parallel programmer simulation professionals and there exists a need for automatic parallelization mechanisms [Fujimoto, 1993].

Automated parallelization is difficult when existing synchronization mechanisms are utilized. In fact, currently there is no published synchronization framework that is readily capable of providing a turn-key solution with any commercial off-the-shelf (COTS) simulation package. There are many issues involved such as modular integration of synchronization mechanisms into existing COTS simulation software, automated development of synchronization infrastructure and automated deployment of turnkey distributed simulation systems.

Automation of parallelization of distributed simulations forms one of the major contributions of this research. Since time-stepped simulation synchronization isolates the actual execution of logical processes from the control mechanism that synchronizes the logical processes, it becomes possible to create a system for automated synthesis and integration of the control mechanism used for synchronization.
The adaptive time-stepped synchronization framework presented in this dissertation is the first formal approach to provide a synchronization technique with automated integration capability with a COTS simulation package with no additional coding required. This capability addresses the issue of rapid development and deployment of distributed simulations with little knowledge of parallel simulation systems.
Chapter 2

Background

In this chapter, a review of the parallel and distributed simulation (PADS) literature is presented with emphasis on synchronization techniques. The chapter is organized as follows. Section 2.1 provides a brief history of the parallel and distributed simulation. Section 2.2 discusses conservative synchronization protocols. Section 2.3 presents optimistic synchronization techniques. Finally, Section 2.4 discusses the research on adaptive/hybrid synchronization mechanisms.

2.1 Brief History of Parallel and Distributed Simulation

Most of the work on the parallel and distributed simulation (referred to as PADS) systems has been initiated by military applications. Battlefield simulations provide a cost efficient way of training military personnel and therefore have been utilized at many different areas of the military. Extensive use of training simulations and the need for creating close-to-real war scenarios has created the need for combining multiple simulations developed by different military units at different locations.

The first work done in the United States, for military applications, was named SIMNET (Simulator Networking). SIMNET was used to connect autonomous simulators for training purposes. Later, building on the technology encapsulated in SIMNET, a distributed simulation environment called DIS (Distributed Interactive Simulation) was developed, in which standards were defined for interconnecting independently developed training simulators in geographically distributed simulation environments [Baker, 1999].

Another development initiated from SIMNET by the U.S. Department of Defense was the ALSP (Aggregate Level Simulation Protocol), which targeted the interconnection
of higher-level war game simulations to create large exercises for joint military operations. Although both DIS and ALSP were successful applications of distributed simulation systems, they did not define a basic standard for creating simulations that are later connected with other simulations to run in a distributed environment [Baker, 1999].

In 1995, U.S. Department of Defense started a new project for the development of the baseline architecture for distributed simulation. This new initiative was named the High Level Architecture (HLA), main objective of which was to establish a common high-level simulation architecture to facilitate the interoperability of all types of models and simulations that interact among themselves.

HLA has emerged from DIS and ALSP to create a standard architecture for adoption by any kind of simulation model or simulator independent of its application level and detail. HLA is currently the standard mandated by the U.S. Department of Defense for military simulations. It has also been adopted by commercial simulation products (such as Arena™ of Rockwell Automation and Flexsim™ of Flexsim Software Products, Inc.) and an IEEE standard has been written for HLA applications [Baker, 1999].

Along with the U.S. military efforts for creating distributed simulation environments, the Australian Defense Science and Technology Organization (DSTO) has been designing and developing distributed simulation applications for military command, control, communication and intelligence (C3I) systems. Distributed Interactive C3I Effectiveness (DICE) simulation, Distributed Generic Agent Modeling Environment (dGAME) and The Integrated Avionics System Support (IASS) simulations are among the efforts initiated by Australian military applications [Baker, 1999].

Although the main thread of distributed simulation research has been developed by military organizations, non-military applications in the computer gaming industry have emerged. Development of fantasy role-playing (FRP) games with distributed virtual
environments captured the attention of millions of computer gamers around the world. Internet gaming option is now available in many games where multiple gamers around the world can participate in virtual competitions. Availability of broadband internet connections enables gamers to enjoy uninterrupted multiplayer gaming over extended periods.

While distributed multiplayer gaming applications attract intense development activities in this field, distributed simulations of business collaborations and supply chain systems are becoming the focus of commercial efforts in this area. In recent years, major companies established overseas manufacturing facilities in order to reduce production costs. Companies around the world are creating business alliances to create services and goods that have higher quality and less production cost. Geographically distributed business collaborations force companies to find ways of analyzing the efficiency of these highly complicated interacting systems of systems.

Industrial simulations have been extensively used to analyze the performance of manufacturing and service systems. However, when it comes to simulating a supply chain system composed of multiple factories and warehouses around the world connected by several transportation systems, development of a single high-fidelity simulation model of the whole system is almost impossible. This is due to many reasons including the increased complexity of the overall system, protection of the proprietary information of the individual companies taking part in the supply chain and limited accessibility to the resources required to create a single simulation model capturing the information embodied by the distributed business entities.

Currently, there are a few companies providing services and software products for the development of distributed simulation systems (a list of these companies are provided at [DMSO]). These are mostly small to medium sized companies providing military simulation solutions. At this time, there are not many publicly available commercial
applications for distributed simulation. This is most probably due to the lack of integrated solution systems that can be rapidly deployed at large scales.

However, research towards implementing distributed simulations of large-scale supply chains is advancing rapidly. Barnett and Miller [2000] developed a modeling and simulation environment called e-SCOR, which is built on the HLA infrastructure and the Supply Chain Operations Reference (SCOR) model developed by The Supply Chain Council [SCOR]. Individual simulation processes developed using e-SCOR environment implement a conservative time management method that requests simulation time advances from the HLA’s Run Time Infrastructure (RTI) and make the simulation process wait until the time advance is granted by RTI.

Sudra et al. [2000] developed a distributed environment called GRIDS (Generic Runtime Infrastructure for Distributed Simulation) that made use of agents responsible for the synchronization of the distributed simulation process. GRIDS was implemented on a virtual supply chain scenario that was earlier developed in [Archibald et al., 1999].

The problem of synchronization has been addressed extensively in the PADS literature. Two major approaches were developed: conservative and optimistic. Current research is moving towards adaptive/hybrid synchronization protocols, which combine the best properties of both conservative, and optimistic approaches with an adaptive decision making process.

### 2.2 Conservative Synchronization Mechanisms

Conservative synchronization protocols were the first ones to be developed in the history of distributed simulation research. The idea is very simple; a conservative synchronization protocol ensures that no simulation process receives a message that schedules an event with a timestamp less than the local clock of the simulation process. This is done by controlling the execution of the events at each local simulation process
and not allowing the execution of any event unless it is impossible to receive an external event (a message scheduling an event) with a timestamp less than the timestamp of the current local event to be executed.

Assume that each simulation process has multiple channels for receiving messages. Each channel corresponds to a link to another simulation process. Also, assume that the messages are received in the same order that they were sent on a single channel. The local simulation process maintains a channel clock for each channel. At each channel, received messages are ordered in increasing timestamp order and messages are processed starting from the top of the channel. The channel clock is always kept set to the minimum timestamp of all of the messages waiting to be processed at that channel. If there are no messages waiting to be processed in a channel, then the channel clock remains set to the timestamp of the last processed message. A simulation process calculates the minimum of all channel clocks before executing any events. If the minimum channel clock value is greater than the timestamp of the next event, the event is safe to be processed. Since the messages in the channels are processed in increasing timestamp order and all messages are received in the order that they were sent, it is not possible to receive a message that has a timestamp less than the minimum channel clock value. This is the main mechanism used in conservative synchronization algorithms.

However, a basic problem with this algorithm, if a channel is not active for a long time, its clock would not be updated and will keep the minimum channel clock value fixed. In this case, the simulation model will be blocked after a while and will not be able to continue executing until a message is received in the blocking channel. Moreover, a cycle of blocking simulation processes might be formed, waiting each other to send a message in the blocking channels. This is known as a deadlock. This problem is addressed by using null messages. Each simulation process calculates a look-ahead value for each of its outgoing message channels and sends it to the connected simulation processes. Look-ahead value is a measure of a safe period on that channel, during when no messages will be send by the source simulation process. When a look-ahead value is
received in a channel, the target simulation process updates the channel clock to the look-ahead value and the simulation execution goes on. This way, although there are no messages in a channel, null messages carrying look-ahead information can update the channel clock.

This null message algorithm is the first conservative synchronization algorithm independently developed by Chandy and Misra [1979] and Bryant [1977] and known as Chandy-Misra-Bryant (CMB) algorithm. Calculation of the look-ahead value is the single most important factor in the CMB algorithm that significantly affects the performance of the algorithm. The reason is that, if the look-ahead values are too small, null messages will be required more frequently and this will create a communication overhead. [Fujimoto, 2000] demonstrates the importance of the efficient calculation of the look-ahead using experimental analysis and points out that, look-ahead values are model dependent and if a simulation model has bad look-ahead characteristics, then the increased amount of messaging overhead may undermine the performance of the CMB algorithm.

An alternative way to deal with deadlocks was proposed by Chandy and Misra [1981]. In this approach, instead of avoiding the deadlocks, the simulation is allowed to enter a deadlock and then the deadlock is resolved by processing the smallest time stamped event in the entire distributed system. However, for this approach to work, the algorithm should be capable of detecting a deadlock and locating the event with the smallest timestamp.

Vee and Wen-Jing [1999] argued that although the deadlock resolution algorithm is superior to the previous CMB algorithm, it creates too many sequential executions if the simulation system is prone to deadlocks. They stated that deadlocks could occur frequently when the number of messages in a system is relatively lower compared to the number of channels.
Another class of conservative algorithms is time-barrier synchronization. In this technique, a time-barrier is set in the simulated time and at this time all simulation processes stop and cooperate to determine the safe events to process and then process them. There are several versions of barrier synchronization and an in-depth review is provided in [Fujimoto, 2000]. One important issue that is addressed by barrier synchronization is the distance between the simulation processes. Distance is the measure of how soon an event in one simulation process can affect another event at a remote simulation process. This is similar to the “virtual interaction lag” (VIL) measure used in this research.

Conservative synchronization mechanisms are easier to implement with respect to optimistic approaches. However, their performance greatly relies on some parameters (such as look-ahead) specific to the simulation models being used. Therefore, conservative approaches are mostly blamed to be inconsistent and several attempts have been made to customize the conservative methods for specific applications [Vee and Wen-Jing, 1999].

As discussed later in this chapter, adaptive mechanisms were developed to provide solutions to this problem.

### 2.3 Optimistic Synchronization Mechanisms

Optimistic synchronization mechanisms have a very different philosophy compared to the conservative mechanisms: In optimistic simulation, the individual simulation processes do not always need to be synchronized. While conservative techniques try to keep the simulation always synchronized by executing only safe events, optimistic methods allow the simulations to execute out of order. However, time ambiguities are detected in a timely manner and any error in causality is resolved by rolling back the execution of the affected simulation processes. In this respect, optimistic mechanisms can be considered as reactive approaches to synchronization.
Eight years after the development of the first conservative CMB algorithm, Jefferson developed the idea for the first and the most well known optimistic approach called the “Time Warp” [Jefferson and Sowizral, 1985]. In time warp, the events being processed in the simulation processes are not discarded but saved for possible future rollbacks. When a rollback occurs, the events that have been executed after the causality error are cancelled and if any of these events have sent messages to other processes, anti-messages are sent to rollback these remote events. This way, a wave of rollbacks propagates through the affected simulation processes and the entire simulation is rolled back to the state before the error occurred. This approach tries exploiting the inherent independence of loosely coupled simulation processes that have infrequent interactions by being optimistic.

An immediate problem with this approach is memory management. If simulation processes save all events during the entire course of execution, the amount of memory available (such as hard drive storage space or system memory) in the processors may not be enough. Therefore, a fossil collection mechanism was developed for releasing memory from obsolete events. In order to detect these obsolete events, a lower bound on the extent of rollbacks is calculated. This lower bound is named the Global Virtual Time (GVT).

GVT is calculated by taking a snapshot of the entire simulation system and identifying the message (this can be an unprocessed message, a message in transit or an anti-message) that has the earliest timestamp. Once GVT is calculated, all of the events with timestamps strictly less than the GVT can be discarded. Performance of time warp depends on the fossil collection mechanism in complex simulation systems and therefore calculation of a good GVT is important. For this reason, many researchers have addressed the GVT calculation problem in the context of time warp. A collection of the research on the GVT calculation is provided in [Vee and Wen-Jing 1999] and [Fujimoto, 2000].
Time warp became the most researched synchronization technique over the past few years and many extensions have been proposed. [Gafni et al., 1988] proposed a rollback mechanism in which the anti-messages are not sent immediately, but the simulation process determines if an anti-message should be sent after re-executing the events beyond the rollback time (some events may be re-executed resulting in sending of the same messages). This technique is termed “lazy cancellation”.

Another similar rollback technique, “lazy reevaluation” was proposed by West [1988]. In this technique, the events are not cancelled immediately after a rollback, but the simulation process determines and cancels only the non-recurring events. Although both lazy cancellation and lazy reevaluation techniques save processing time during rollback, they require an extra amount of memory for storing events and decision-making.

One of the most well known implementations of the optimistic approach is called the Georgia Tech Time Warp (GTW), which is described in detail in [Fujimoto, 2000]. GTW combines multiple techniques and defines a unifying synchronization executive. GTW has been implemented for the distributed simulation of wireless networks, air traffic systems and ATM (asynchronous transfer mode) networks. It has been reported in [Fujimoto, 2000] that GTW was able simulate up to 28,000 events per second using only two processors in a shared memory multi-processor architecture (Kendall Square Research KSR-2).

Other optimistic synchronization mechanisms have been proposed in the literature. A moving time window algorithm was developed by Sokol and Stucky [1990] in which the amount of gap between the local clocks of the simulation processes are bounded by a certain value. This technique prevents some of the fast processes to advance too far ahead of others creating potential for erroneous interactions.
A breathing time buckets (BTB) algorithm was developed by Steinman [1993]. In BTB, an event-horizon is calculated by the processes to determine the size of a time window that is used in combination with local rollbacks and time barriers to regulate advance of the local simulation clocks.

Another advanced optimistic method is called Wolf Calls and developed by Madisetti et al. [1988] in which a special control message is used to stop advancing of the simulation processes once a roll-back is detected in one of the processes. This limits the spread of an erroneous execution and therefore the depth of the rollback.

Optimistic synchronization mechanisms have been more extensively researched than other synchronization mechanisms, and many advanced techniques governing several details of the time warp based execution have been developed. However, memory requirements and additional overhead associated with frequent rollbacks in “not so loosely coupled” systems create criticism about the performance of the optimistic protocols. For this reason, a new research direction has been pursued for the development of adaptive techniques, which adjust the parameters of the synchronization mechanisms during runtime according to the state of the simulations.

2.4 Adaptive Synchronization Mechanisms

Many simulation models behave dynamically during their executions. The interactions between simulations can occur randomly and these interactions should be processed properly for synchronization conditions to be satisfied. Therefore, the synchronization mechanism must adapt itself to changes in the interactions during the execution of a system. Usefulness of adaptive synchronization was first reported by Reynolds [1988] in which the adaptability is defined as the ability of the simulation processes to change selected control variables dynamically based on the partial state information of the distributed simulation.
Adaptive protocols appeared in the literature are based on the idea of dynamically adjusting the degree of conservative and optimistic behavior by examining the state of the simulation. An early attempt in this field was a modification of the time warp mechanism by Reiher et al. [1989]. A penalty based throttling technique was used to adjust the degree of optimism in time warp by penalizing the simulation processes that had frequent rollbacks in the recent past of the simulation.

Ball and Hoyt [1990] developed the Adaptive Time Warp (ATW) algorithm in which simulation processes are blocked between processing successive events based on the statistical estimates of time between messages received from other simulation processes.

Another similar work was proposed by Ferscha [1995], which was named as the PADOC protocol (Probabilistic Adaptive Direct Optimism Control). Ferscha and Johnson [1999] proposed a proactive method called the “Shock Resistant Time Warp” in which the future state of the simulation is predicted based on the past behavior and proper optimism level is selected by limiting the memory available to time warp. Panesar and Fujimoto [1997] proposed a flow control mechanism in time warp, which dynamically determines a window of uncommitted events for execution. The window size is dynamically adjusted based on the execution characteristics of the simulation.

While all of the previously mentioned protocols depend on the partial state information of the simulation, a few adaptive protocols relying on the global state of the simulation was also developed. Das et al. [1994 and 1997] proposed the “Adaptive Memory Management” protocol, which controls the optimism of time warp by managing the available memory to the simulation kernel. Extensive feedback from the simulation system such as fossil collection frequency, event cancellation and rollback behavior is examined to determine the optimum memory level.
Srinivasan and Reynolds [1995 and 1998] proposed the “Elastic Time” algorithm based on what is called the “Near Perfect State Information” (NPSI). NPSI protocols calculate a measure of risk, called the error potential (EP), for each simulation process. EP is used to control the degree of optimism during execution. Elastic Time algorithm limits the gap between the local clocks of the simulation processes using dynamically calculated EP values. In a way, the risk of facing a time ambiguity is reduced by limiting the advance of simulation processes.

A detailed survey and critique of the adaptive synchronization protocols is presented in [Das, 2000]. In this work, Das suggested an interesting point: all of the adaptive synchronization mechanisms that have appeared in the literature are based on the time warp protocol. Research of adaptive synchronization based on solely conservative techniques is an open area and this research developed one such technique inspired by real-time simulation of distributed systems.

A summary of the major synchronization protocols discussed in this chapter is provided in Table 2-1.
### Table 2-1: Major Synchronization Protocols

<table>
<thead>
<tr>
<th>Class</th>
<th>Protocol</th>
<th>Basic Description</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conservative</td>
<td>Chandy-Misra-Bryant (CMB) Algorithm</td>
<td>A lookahead is used to determine safe events to execute and synchronization is maintained continuously by executing only the safe events in incoming message channels of simulation processes.</td>
<td>Chandy and Misra [1979]; Bryant [1977]</td>
</tr>
<tr>
<td></td>
<td>Time Warp</td>
<td>Simulations execute in parallel until a time-ambiguity occurs and then faulty simulations roll-back to correct the causality violations.</td>
<td>Jefferson and Sowizral [1985]</td>
</tr>
<tr>
<td></td>
<td>Time Warp with Lazy Cancellation</td>
<td>Roll-backs are propagated only after the simulation process where the causality violation first occurred re-executed beyond roll-back time.</td>
<td>Gafni et al. [1988]</td>
</tr>
<tr>
<td></td>
<td>Time Warp with Lazy Reevaluation</td>
<td>Only the non-recurring events are cancelled during a roll-back.</td>
<td>West [1988]</td>
</tr>
<tr>
<td></td>
<td>Georgia Tech Time Warp (GTW)</td>
<td>A well-known implementation of the Time Warp mechanism that combines multiple extensions and provides a unifying synchronization executive.</td>
<td>Fujimoto [2000]</td>
</tr>
<tr>
<td></td>
<td>Moving Time Window Algorithm</td>
<td>The amount of gap between local clocks of simulation processes are bound by a fixed value.</td>
<td>Sokol and Stucky [1990]</td>
</tr>
<tr>
<td></td>
<td>Breathing Time Buckets (BTB) Algorithm</td>
<td>An event-horizon is calculated to determine the size of a time window to be used in combination with local roll-backs and time barriers to regulate advance of local simulation clocks.</td>
<td>Steinman [1993]</td>
</tr>
<tr>
<td></td>
<td>Wolf Calls</td>
<td>A special control message is used to stop advancing of simulation processes once a roll-back is detected in one of the processes.</td>
<td>Madisetti et al. [1988]</td>
</tr>
<tr>
<td>Optimistic</td>
<td>Penalty Based Throttling</td>
<td>Adjusts the degree of optimism in the Time Warp mechanism by penalizing simulation processes with frequent roll-backs.</td>
<td>Reiher et al. [1989]</td>
</tr>
<tr>
<td></td>
<td>Adaptive Time Warp (ATW)</td>
<td>Simulation processes are blocked between successive events based on statistical estimates of time between messages received from other simulation processes.</td>
<td>Ball and Hoyt [1990]</td>
</tr>
<tr>
<td>Adaptive</td>
<td>Shock Resistant Time Warp</td>
<td>Future state of the simulation is predicted based on the past behavior and proper optimism level is selected by limiting the memory available to Time Warp.</td>
<td>Ferscha and Johnson [1995]</td>
</tr>
<tr>
<td></td>
<td>Adaptive Memory Management</td>
<td>Controls the optimism of Time Warp by managing the available memory to the simulation kernel.</td>
<td>Das et al. [1994 and 1997]</td>
</tr>
<tr>
<td></td>
<td>Elastic Time</td>
<td>Calculates a measure of risk based on Near Perfect State Information (NPSI) which is used to control the degree of optimism during execution by limiting the gap between local clocks dynamically.</td>
<td>Srinivasan and Reynolds [1995 and 1998]</td>
</tr>
</tbody>
</table>
Chapter 3

Structural Models and Analyses of Synchronization Requirements

A discrete event simulation process, whether executed individually or as part of a distributed simulation system (DSS), can be represented mathematically using state machines, i.e., a state automaton. In general, a state automaton is a mathematical representation of the dynamics of discrete state space processes. There are several types of state automata, which form the fundamental theory of computation [Hopcroft et al., 2001].

In this chapter, a state automaton representation of discrete event simulation processes is introduced with formal definitions. A basic state automaton model is discussed followed by a messaging and communication procedure and finally the timing structure for such a system. This modeling framework is used to analyze the property of synchronization in DSSs. The model representation provides a generic vision of discrete simulation systems and provides the key linkages for simulation coordination.

3.1 Basic Structural Model

In order to demonstrate the modeling approach the following example is provided.

Example 3.1 Consider a queuing system with two identical servers and one arrival process. Parts arrive in this system at a constant rate and are processed by either of the servers. Figure 3.1 shows this system.
In order to create a simulation model of this system, several parameters need to be defined. Among these parameters are:

- Arrival process parameters
- Queue discipline
- Parameters of service process
- Part types

These parameters define the dynamics of the system. In particular, the queuing discipline is essential since it defines how parts are routed in the system.

A simulation process can be decomposed into two fundamental components: (1) a structural model, and (2) the flow logic and data. The structural model statically defines the system being simulated. Figure 3-1 is a pictorial representation of the structural model for the example system. The simulation data characterizes the specific behavior of the components in the structural model while the flow logic defines the dynamics of the system. This decomposition of simulation processes provides a valuable way of creating an abstraction of a simulation process for the purposes of generic investigation.

The concept of a structural model is similar to a connectivity graph (or a physical graph as defined by [Mettala, 1989]) which shows the possible routes of flow of entities among system resources. The connectivity graph for example 3.1 is given in Figure 3-2. Nodes represent system resources and arcs represent flow of entities.

Figure 3-1: A simple 2-server queuing system
An entity is a dynamic object which moves through the simulation process and is processed (e.g., a part in a manufacturing system or a packet in a communication network or a print job running on a print server).

A resource is an object that is utilized by the entities for a finite duration of time. Resources can be active objects that act on the entity and change its state and/or properties (e.g., machines processing a part in a job shop, processors of a computer calculating an output on a given input, etc.). There are also passive resources which do not change the status of an entity but store them while they are waiting for active resources to become available (e.g., buffers in assembly stations, print queues in a print server, etc.).

A policy is a set of rules that defines the path of entities in the system (e.g., scheduling policy in a job shop). These policies are captured in the flow logic of simulation model.

The structural model of a system only describes system resources and their accessibility by entities. It does not provide any information about the policy of the system, which makes it simpler in form. In this research, a specific form of a structural
model called the “entity-flow model” is used, which is based on state automata with input/output functionality and a virtual timing structure. The basic automaton model will first be discussed.

Let $M$ be a simulation model defined by the partition $(S | P | R | E)$ where $S$ is a state automaton representing the structural model, $P$ is a decision function representing the flow logic (and data embedded within), $R$ is a resource set (including resource states and attributes) and $E$ is the entity set.

The decision function $P : E \times R \times X \times \Gamma \rightarrow \Sigma$ is a mapping, based on the status of resources, from the set of enabled events at a specified state of an entity to a single selected event (selected transition). This function selects among alternative transitions for an entity (it represents a policy). It is not important to elaborate on the decision function since our modeling approach excludes policy information.

$S$ is a state automaton defined by the 5-tuple $S = (X, \Sigma, \Gamma, \Psi, \overline{X})$, where

- $X$ is a countable state space, such that $X \neq \emptyset$
- $\Sigma$ is a countable event set, such that $\Sigma \neq \emptyset$
- $\Gamma : X \times E \rightarrow \Sigma$ is the set of enabled events for all entities $e \in E$ at all states $\chi \in X$.
- $\Psi : X \times \Sigma \rightarrow X$ is the transition mapping defined for all states in $X$ and all enabled events in $\Sigma$.
- $\overline{X} \subseteq X$ is the set of initial states where an entity starts its flow.

The structural model presented here is different from a typical state machine representation of a discrete event system. In general, a state machine model of a physical system embodies system variables representing resource states. The structural model presented here only contains states of the entities and disregards resource states since events or state transitions are the focus of coordination.
Next, an entity specific reduction is defined for the general structural model and shows its relation to the general structural model.

For each entity $\varepsilon \in \Sigma$ the 4-tuple $S_\varepsilon = (X_\varepsilon, \Sigma_\varepsilon, \Psi_\varepsilon, \overline{X_\varepsilon})$ is defined to be an entity-specific structural model of $M$, where

- $\bigcup_{\varepsilon \in \Sigma} X_\varepsilon = X$
- $\bigcup_{\varepsilon \in \Sigma} \Sigma_\varepsilon = \Sigma$
- $\bigcup_{\varepsilon \in \Sigma} \Psi_\varepsilon(\chi, \sigma) = \Psi(\chi, \sigma), \forall \chi \in X, \forall \sigma \in \Gamma(\chi)$
- $\bigcup_{\varepsilon \in \Sigma} \overline{X_\varepsilon} = \overline{X}$

The entity-specific structural model is a reduced version of the general structural model simplified by including only necessary states, events and transitions used by an entity (as defined by the function $\Gamma$ in the general structural model). The composition of entity-specific structural models forms the general structural model. Next, we provide an example scenario to demonstrate synthesis of structural models.

**Example 3.2** Consider the queuing system shown in Figure 3.1. It is necessary to define part specifics for this system. There are two part types for this case: type A and type B. A part of type A arrives in the system every 6 minutes and a part of type B arrives every 4 minutes. Type A can only use server 1 while type B can use any of the two servers. The processing time for type A parts is 2 minutes and for type B parts is 4 minutes. The parts arrive and if the servers are busy they enter the queue. When a server becomes available the earliest arrived part that has the minimum processing time on that server is processed next.

The general structural model for this system is defined as follows:
\( X = \{A, Q, S_1, S_2, D\} \)

\( \Sigma = \{\text{arr, serv}_1, \text{serv}_2, \text{dep}\} \)

\( \Gamma(A) = \{\text{arr}\} \)

\( \Psi(A, \text{arr}) = Q \)

\( \Gamma(Q) = \{\text{serv}_1, \text{serv}_2\} \)

\( \Psi(Q, \text{serv}_1) = S_1 \)

\( \Gamma(S_1) = \{\text{dep}\} \)

\( \Psi(Q, \text{serv}_2) = S_2 \)

\( \Gamma(S_2) = \{\text{dep}\} \)

\( \Psi(S_1, \text{dep}) = D \)

\( \Gamma(D) = \emptyset \)

\( \Psi(S_2, \text{dep}) = D \)

\( \overline{X} = \{A\} \)

Being a state machine, the structural model can be represented by a directed graph. This graph is similar to the physical graph but it includes all of the information embedded in the automaton. The directed graph for the above defined structural model is given in Figure 3-3.

![Directed graph representation of the structural model](image)

**Figure 3-3**: Directed graph representation of the structural model

In this directed graph, nodes represent states and arcs represent transitions. Each arc has a label corresponding to the event that occurs during that transition. The node representing state \( A \) is marked on the right to show that it is an initial state (i.e. a member of \( \overline{X} \)). Notice that \( D \) is a terminal or absorbing state which represents the end of simulation for an entity.

There are two entity-specific structural models for this system since there are two types of entities. The entity set for this system is defined as \( E = \{\text{ent}_A, \text{ent}_B\} \). \( S_{\text{ent}_A} \) is the same as the general structural model since entity type B has access to all system
resources. However, entities of type A cannot visit server 1. Therefore, $S_{\text{ent}_A}$ is different from the general model as shown below:

\[
\begin{align*}
X_A &= \{A, Q, S_2, D\} \\
\Gamma_A(A) &= \{\text{arr}\} \\
\Gamma_A(Q) &= \{\text{serv}_2\} \\
\Gamma_A(S_2) &= \{\text{dep}\} \\
\Gamma_A(D) &= \emptyset \\
\bar{X}_A &= \{A\}
\end{align*}
\]

\[
\begin{align*}
\Sigma_A &= \{\text{arr}, \text{serv}_2, \text{dep}\} \\
\Psi_A(A, \text{arr}) &= Q \\
\Psi_A(Q, \text{serv}_2) &= S_2 \\
\Psi_A(S_2, \text{dep}) &= D
\end{align*}
\]

The graphical representation for $S_{\text{ent}_A}$ is given in Figure 3.4 and it is a reduced form of the general model’s graph given in Figure 3.3. By hiding the unrelated states and events, it is possible to reduce the size of the automaton and eliminate unnecessary complexity. This idea is especially useful for both modeling and implementation purposes.

![Figure 3-4: Directed graph representation of $S_{\text{ent}_A}$](image)

Structural models have a simple form since they do not include any decision logic. The decision function (which corresponds to the policy of the system) determines which entity goes through which transition when a resource becomes available. The simulation model has built-in variables (such as the status of the servers, arrival times of entities etc.) and embedded logic to route entities within the system model. A Petri-net model, for instance, can be created to include all of these details, and the simulation process can be completely represented. However, the complexity increases substantially when decision logic is included in a model. Most of the effort spent while modeling a
physical system is focused on accurately capturing the policy of the system in the flow logic of the simulation model. By excluding the flow logic from the model, it is possible to easily synthesize less complex structural models of the underlying system, and it is possible to use structural models of simulation processes to analyze their behavior in a distributed setting.

Before a model of a simulation process within a DSS can be completed, two more additions to the model definition given above are necessary. These are: (1) a messaging mechanism since all the simulation processes in a DSS will exchange information with each other, and (2) a timing mechanism. The timing mechanism is crucial since one needs to know when particular events occur during the simulation process (i.e., message sending events).

The messaging capability is provided by partitioning the event set into three subsets: input events, output events and internal events. These are defined as follows:

\[ \Sigma = \Sigma_{\text{Input}} \cup \Sigma_{\text{Output}} \cup \Sigma_{\text{Internal}} \]

where:

- \( \Sigma_{\text{Input}} \) is the set of input events (i.e. the events that are executed/scheduled by the receipt of a message from another simulation process)
- \( \Sigma_{\text{Output}} \) is the set of output events (i.e. events that send a message to another simulation process upon execution)
- \( \Sigma_{\text{Internal}} \) is the set of internal events (i.e. events that neither receive nor send a message)

Similarly, the event sets of the entity-specific models are also partitioned. Each output event has a corresponding input event; however these events occur at different simulation processes, namely a sender and receiver. A simulation process can be a sender, a receiver or both. The union of input and output events are named as external events (\( \Sigma_{\text{Input}} \cup \Sigma_{\text{Output}} = \Sigma_{\text{External}} \)). Any event that is not an external event is named as an
internal event \( (\Sigma_{\text{External}} \cap \Sigma_{\text{Internal}} = \emptyset) \). Any simulation process in a DSS must have a non-empty external event set \( (\Sigma_{\text{External}} \neq \emptyset) \). Output events are of particular interest, because execution of an output event creates a possible disturbance of the synchronization property. Timing of output events is the single most important information that we need when analyzing the synchronization characteristics of a DSS.

A bound-map that represents the maximum and minimum amount of virtual time an entity can spend in a state is defined. As discussed earlier, states correspond to resources in the system and entities utilize resources for certain amounts of virtual time during an execution. Most simulations are stochastic and delays are defined as random variables sampled from pre-determined probability distributions. Although in theory some probability distributions can generate unbounded values (such as the Normal distribution), it is impractical for a process to take an infinite amount of time. For this reason, it is assumed that all probability distributions used in the simulation models generate random values with known lower and upper bounds. In fact, realistic simulation implementations use empirical distributions which model stochastic nature of real-life processes more accurately. Nevertheless, in many cases one can use truncated versions of unbounded probability distributions without disturbing underlying modeling assumptions.

The bound-map \( B \) simply assigns two virtual time values, a lower bound \( l_x \) and an upper bound \( u_x \) to each state \( \chi \in X \), where \( 0 \leq l_x \leq u_x < \infty \). Thus, \( B \) is defined as \( B : X \rightarrow \mathbb{Z}^+ \times \mathbb{Z}^+ \). For each entity-specific model \( S_\epsilon \), an entity-specific bound-map is defined as \( B_\epsilon : X_\epsilon \rightarrow \mathbb{Z}^+ \times \mathbb{Z}^+ \), where a lower bound \( l^\epsilon_x \) and an upper bound \( u^\epsilon_x \) are defined for each state \( \chi \in X_\epsilon \). The relation between the bound-maps of a structural model and entity-specific models are given by:

\[
l_x = \min_{\forall \chi \in E} \{ l^\epsilon_x : \chi \in X_\epsilon \} \text{ and } u_x = \max_{\forall \chi \in E} \{ u^\epsilon_x : \chi \in X_\epsilon \}, \forall \chi \in X
\]  

(3.1)
Bound-maps for entity-specific models can be extracted from the simulation model by examining the processing times of entities. However, some resources such as queues do not have well-defined time bounds. For these passive resources, the lower bound is always zero (e.g., no waiting time in a queue) and the upper bound can be assigned a very big number (e.g. total runtime of the simulation). This issue does not disturb the utility of the model since upper bounds for these resources are not of importance in our analysis. Specifically, lower bounds will be used to calculate earliest occurrence times of output events.

Using the virtual clock structure and the bound-map, a simulation run or an execution can be defined. An execution \( r = \{T_0, s_0, e_1, T_1, s_1, e_2, \ldots, T_n, s_n\} \) of a simulation process is an ordered set of states, events and corresponding virtual clock values where \( T_0 \leq T_1 \leq T_2 \leq \ldots \leq T_n \) and \( n \in \mathbb{Z}^+ \). \( T_n \) is the terminal LVT value of the simulation. Each event corresponds to a transition to the next state in the ordered set. Again, it is assumed that any execution of a simulation has finite number of transitions and since \( T_k - T_{k-1} \leq u_{\text{s}} \), \( \forall k \leq n \) we have \( T_n < \infty \), that is, a simulation must have a finite runtime.

With the addition of the bound-map the general structural model takes the form \( S = (X, \Sigma, \Gamma, \Psi, \bar{X}, B) \) and the entity specific model becomes \( S_\varepsilon = (X_\varepsilon, \Sigma_\varepsilon, \Psi_\varepsilon, \bar{X}_\varepsilon, B_\varepsilon) \).

In order to simplify the notation (since multiple structural models will be considered while analyzing the properties of a DSS), a dot-notation of object oriented programming is used and is defined as follows:

- A structural model \( S \) of a logical process \( LP_k \) is denoted by \( LP_k \cdot S \)
- A component \( C \) of a structural model is denoted by \( S.C \)

For example, \( LP_k \cdot S_\varepsilon \cdot \Sigma_{\text{Output}} \) denotes the output event set of the \( \varepsilon \)-specific structural model of the logical process \( LP_k \).
The general structural model defined in this chapter forms the basic tool for analyzing the synchronization properties and requirements for a DSS. Although the model developed here can be represented as a push-down automaton, the purpose of use of the model deserves a more explanatory name. Therefore, this model is named, in accordance with the existing literature, a communicating timed state automaton.

A DSS can be modeled as a composition of structural models belonging to constituent simulation processes. This provides a formal modeling environment suitable for analyzing timing properties of output events and developing the sufficient conditions for synchronization.

### 3.2 DSS Structural Model

The structural model for the DSS is formed by combining the structural models of the constituent logical processes. The entity specific structural model for each LP was defined. The entity specific structural model is a reduction of the general structural model for a type of entity. The definition of an entity set for each simulation model was also discussed. The union of all the entity sets basically forms the entity set for the DSS. Entity sets do not contain all of the entities of a simulation, but they are formed of the entity classes (or entity types). Each entity flowing in the simulation is an instance of a certain entity class in the entity set. In a DSS, all logical processes might have the same entity set (i.e., any entity can reach any simulation model). However, it is also likely that some entity types never visit some logical processes. Therefore, entity sets of logical processes need not be the same as the entity set of the DSS.

Entity specific structural models of LPs can be combined (for each entity type) to create the entity specific structural models for the DSS. The flow of entities within the DSS is represented by messages passed between the LPs. An output event corresponds to the departure of an entity while the matching input event corresponding to the arrival of
the entity at the destination LP. The combination of the entity specific structural models can be done by replacing every pair of output and input events with a special transition (transportation event) while also combining the final and initial states. An example combination is given in Figure 3.5.

Figure 3-5: Combination of two entity specific models

Once, all the entity specific structural models are defined for the DSS, these models can be used to form the general structural model for the DSS (the relation between the entity specific models and the general model was defined in the previous section).

First, a “compatibility condition” for the set of general and entity specific structural models is defined. Compatibility simply means that two structural models can be combined.

Consider two logical processes $LP_1$ and $LP_2$ with the corresponding entity sets $E_{LP_1}$ and $E_{LP_2}$, respectively. The first condition for compatibility is of course the existence of common entity classes, that is $E_{LP_1} \cap E_{LP_2} \neq \emptyset$. 
For a common entity type \( \varepsilon \in E_{\text{LP}_1} \cap E_{\text{LP}_2} \) the following conditions must be true for \( \text{LP}_1.\varepsilon \) and \( \text{LP}_2.\varepsilon \) to be compatible:

- \( \text{LP}_1.\varepsilon \cdot \Sigma_{\text{External}} \cap \text{LP}_2.\varepsilon \cdot \Sigma_{\text{External}} \neq \emptyset \)
- \( \exists \sigma \in \text{LP}_1.\varepsilon \cdot \Sigma_{\text{Input}} \land \exists \sigma' \in \text{LP}_2.\varepsilon \cdot \Sigma_{\text{Output}} \) where \( i, j \in \{1, 2\} \) and \( i \neq j \) and \( \sigma = \sigma' \).

The first condition specifies that the entity specific structural models must have common external events and the second condition states that for every output event at the source logical process there exists an input event at the destination logical process with the same event label.

The general structural models of \( \text{LP}_1 \) and \( \text{LP}_2 \) are compatible if \( \text{LP}_1.\varepsilon \) and \( \text{LP}_2.\varepsilon \) are compatible for all common entity types \( \varepsilon \in E_{\text{LP}_1} \cap E_{\text{LP}_2} \), (i.e. all entity specific structural models are compatible for all common entity classes).

Having defined the compatibility property, the operation of “composition” in the set of structural models, for combining entity specific models to form the DSS model becomes critical.

The operation of composition (denoted by \( \oplus \)) is defined for two compatible structural models \( \text{LP}_1.\varepsilon \) and \( \text{LP}_2.\varepsilon \) as follows:

\[
\text{LP}_{[1,2]} \cdot \varepsilon = \text{LP}_1.\varepsilon \oplus \text{LP}_2.\varepsilon
\]

where:

- \( E_{\text{LP}_{[1,2]}} \doteq E_{\text{LP}_1} \cup E_{\text{LP}_2} \)
- \( \forall \varepsilon \in E_{\text{LP}_1} \cap E_{\text{LP}_2} \) the following are defined:
  \[
  LP_{[1,2]} \cdot \varepsilon \cdot \Sigma_{\text{External}} \doteq (LP_1.\varepsilon \cdot \Sigma_{\text{External}} \cup LP_2.\varepsilon \cdot \Sigma_{\text{External}}) - \left( \text{LP}_1.\varepsilon \cdot \Sigma_{\text{External}} \cap \text{LP}_2.\varepsilon \cdot \Sigma_{\text{External}} \right)
  \]
\[ LP_{(1,2)} \cdot S_e \Sigma_{\text{Internal}} \cong (LP_1 \cdot S_e \Sigma_{\text{Input}} \cup LP_2 \cdot S_e \Sigma_{\text{Internal}}) \cup \]
\[ (LP_1 \cdot S_e \Sigma_{\text{External}} \cap LP_2 \cdot S_e \Sigma_{\text{External}}) \]

\[ \forall \sigma \in LP_1 \cdot S_e \Sigma_{\text{External}} \cap LP_2 \cdot S_e \Sigma_{\text{External}} \text{ we have} \]
\[ \sigma \in LP_1 \cdot S_e \Sigma_{\text{Input}} \text{ and } \sigma \in LP_1 \cdot S_e \Sigma_{\text{Output}} \]
where \( i, j \in \{1, 2\} \) and \( i \neq j \)
\[ LP_i \cdot S_e \cdot \Psi(\chi_k, \sigma) = \chi'_k \text{ and } LP_j \cdot S_e \cdot \Psi(\chi_m, \sigma) = \chi'_m \]
where \( \chi_k \in LP_i \cdot S_e \cdot \bar{X} \)

- We define \( LP_{(1,2)} \cdot S_e \cdot \Psi(\chi_k, \sigma) \cong \chi'_k \) and we exclude \( \chi'_k \) and \( \chi_m \) from \( LP_{(1,2)} \cdot S_e \cdot X \). Thus
\[ LP_{(1,2)} \cdot S_e \cdot X = LP_i \cdot S_e \cdot X \cup LP_2 \cdot S_e \cdot X - LP_{(1,2)} \cdot S_e \cdot X_{\text{Excluded}} \]
where \( LP_{(1,2)} \cdot S_e \cdot X_{\text{Excluded}} \) is composed of all excluded states \( \chi'_k \) and \( \chi_m \).

\[ \circ \quad LP_{(1,2)} \cdot S_e \cdot \Sigma_{\text{Input}} \cong LP_1 \cdot S_e \cdot \Sigma_{\text{Input}} \cup LP_2 \cdot S_e \cdot \Sigma_{\text{Input}} - \]
\[ (LP_1 \cdot S_e \cdot \Sigma_{\text{External}} \cap LP_2 \cdot S_e \cdot \Sigma_{\text{External}}) \]

\[ \circ \quad LP_{(1,2)} \cdot S_e \cdot \Sigma_{\text{Output}} \cong LP_1 \cdot S_e \cdot \Sigma_{\text{Output}} \cup LP_2 \cdot S_e \cdot \Sigma_{\text{Output}} - \]
\[ (LP_1 \cdot S_e \cdot \Sigma_{\text{External}} \cap LP_2 \cdot S_e \cdot \Sigma_{\text{External}}) \]

\[ \circ \quad \forall \sigma \in (LP_1 \cdot S_e \cdot \Sigma_{\text{Input}} \cup LP_2 \cdot S_e \cdot \Sigma_{\text{Input}}) - (LP_1 \cdot S_e \cdot \Sigma_{\text{External}} \cap LP_2 \cdot S_e \cdot \Sigma_{\text{External}}) \text{ we have} \]
\[ \sigma \in LP_i \cdot S_e \cdot \Sigma \text{ where } i \in \{1, 2\} \]
\[ LP_i \cdot S_e \cdot \Psi(\chi_i, \sigma) = \chi'_i \text{ where } \chi_i \in LP_i \cdot S_e \cdot X \]

- We define \( LP_{(1,2)} \cdot S_e \cdot \Psi(\chi_i, \sigma) \cong \chi'_i \) where \( \chi_i \in LP_{(1,2)} \cdot S_e \cdot X \)

\[ \circ \quad LP_{(1,2)} \cdot S_e \cdot \bar{X} \cong (LP_1 \cdot S_e \cdot \bar{X} \cup LP_2 \cdot S_e \cdot \bar{X}) - LP_{(1,2)} \cdot S_e \cdot X_{\text{Excluded}} \]

\[ \circ \quad \forall \chi \in LP_{(1,2)} \cdot S_e \cdot X \text{ we define } LP_{(1,2)} \cdot S_e \cdot B(\chi) \cong LP_{(1,2)} \cdot S_e \cdot B(\chi) \]
such that \( \chi \in LP_i \cdot S_e \cdot X \) and \( i \in \{1, 2\} \)

- \( LP_{(1,2)} \cdot S \) is defined by all entity specific structural models \( LP_{(1,2)} \cdot S_e \) such that
\[ e \in E_{LP_{(1,2)}} \]
The definition of composition operation basically states that two compatible general structural models can be combined by combining the entity specific structural models. Entity specific structural models are combined by simply stitching the matching input and output events as depicted in Figure 3-5. Compatibility of models assures that the events for matching input and output transitions are the same. The general structural model of the composition is defined by the constituent entity specific structural models (as defined in the previous section).

Finally, the general structural model for a DSS can be defined. Let \( D =< \overline{LP}, G > \) be a DSS composed of the logical processes \( \overline{LP} = \{ LP_1, \ldots, LP_N \} \) and the connection topology defined by the directed graph \( G = (V, T) \) where \( V \) denotes the set of vertices corresponding to the logical processes and \( T \) denote the set of edges corresponding to the directed links between logical processes. A link from \( LP_1 \) to \( LP_2 \) represents the flow of one or more entity types from \( LP_1 \) to \( LP_2 \). For \( D \) to be a valid DSS, the set of logical processes \( P \) must be compatible, that is, the structural model of every element of \( P \) must be compatible with the structural model of at least one other element of \( P \) which is not itself. Compatibility at this level means that one can combine the structural models of all constituent logical processes to form the structural model of the DSS. Thus, the structural model of \( D \) is defined as \( D.S = LP_1.S \oplus LP_2.S \oplus \ldots \oplus LP_N.S \). This multiple composition operation is performed by pair wise composition of compatible structural models until one combined structural model is left, which is the general structural model for the DSS. The entity set of the DSS, \( D.E \), is the union of the entity sets of the member LPs.

The structural model \( D.S \) is said to be closed if \( D.S \Sigma^\text{External} = \emptyset \), that is, there are no other distributed simulation systems interacting with \( D \). In this study, we will only consider closed distributed simulation systems.
3.3 Timing of Events

Before defining the timing mechanism for the simulation processes, it is necessary to explain the concept of time in more detail. Time is a frame of reference to assist us in organizing or analyzing activities. However, it has been shown that there is no absolute time frame in the universe (i.e., theory of relativity). The term “time”, as we use in our daily lives, actually refers to a standard set on earth, which was named as the “Coordinated Universal Time” or shortly UTC (a language-independent international abbreviation). UTC is a virtual clock time maintained and broadcast by the U.S. Naval Observatory in Washington, D.C. based on sampling and averaging the output of multiple cesium clocks (9,192,631,770 oscillations of the cesium 133 atom exposed to a suitable excitation takes exactly 1 second) in a temperature controlled environment. UTC is broadcast through radio waves and other communication media and used as a standard to calculate local time. Computers around the world periodically sample UTC clock values over the Internet to synchronize their system clocks.

In this study, UTC time is referred to as real-time or the wallclock time. It is the main reference that is used to define certain properties of DSSs. On the other hand, there is another time frame we use in the simulation processes, that is the simulation time. Simulation time need not be synchronized with the real-time. In fact, generally simulations are desired to run faster than real-time. In its common meaning, a simulation is used to analyze a system. However, some simulation processes are used exclusively for controlling physical systems, such as chemical processes or manufacturing facilities. These simulations are designed to run synchronized with real-time.

For generality, the concept of virtual time will be used to refer to values of the clock of a simulation process. Each simulation process has a virtual clock, which is used to organize occurrence of events during execution. In general, virtual time of a simulation process is independent from the real-time. It is simply a counter, which is incremented as events occur during simulation.
In a distributed setting, the virtual time of a simulation process is referred to as local virtual time (LVT) since there is more than one local virtual clock (LVC) in a DSS. The local virtual clock is defined as follows:

\[
LVC : \mathbb{R}^+ \to \mathbb{Z}^+
\]

\[
LVC(t_1) - LVC(t_0) \geq 0 \text{ iff } t_1 \geq t_2 \text{ where } t_1, t_2 \in \mathbb{R}^+
\] (3.2)

\(LVC\) is defined as a mapping of real-time to non-negative integers. Like real-time, value of \(LVC\) always increases. It is important to understand that, in general, \(LVC\) is independent of real-time and its rate of advance is determined by the simulation dynamics. This mapping is defined to create a common time reference within the DSS to analyze the relative ordering of events.

Value of an LVC at any point in real time is called the LVT. For convenience, LVT is defined as a mapping from the set of events \(\Sigma\) to the set of integers, to specify the virtual time of execution of events.

\[
LVT : \Sigma \to \mathbb{Z}^+
\] (3.3)

\(LVC(t_i) = T_i = LVT(\sigma_i)\) means that the virtual time of execution for event \(\sigma_i\) is \(T_i\) and the wallclock time at that instant is \(t_i\).

Figure 3.6 shows advance of an \(LVC\) in a typical event-driven simulation where the \(LVT\) value is updated by the scheduled virtual time of the current event being executed. Execution of each event causes the \(LVT\) to make a positive discrete jump, similar to a step function but with irregular step sizes.
3.4 Event Dynamics and Causal Precedence

The dynamic behavior of a DSS is determined by the execution of events at LPs and the flow of messages between the LPs. An execution of a DSS can be defined as a vector of executions of member LPs. Let $r_i$ denote an execution of $LP_i$, then the execution for the DSS can be defined as $\vec{r} = \langle r_1, ..., r_N \rangle$ where $N$ denotes the number of LPs in the DSS.

Next, the following relations on the set of events $\Sigma$ are defined:

- An event $\sigma_i$ locally precedes another event $\sigma_j$ if and only if event $\sigma_i$ comes immediately before event $\sigma_j$ in an execution. The \textit{locally precedes} relation is denoted as $\sigma_i \prec \sigma_j$.

- An event $\sigma_i$ remotely precedes another event $\sigma_j$ if and only if event $\sigma_i$ is an output event and $\sigma_j$ is the matching input event at a remote LP. The \textit{remotely precedes} relation is denoted as $\sigma_i \xrightarrow{m_y} \sigma_j$ where $m_y$ denotes the message passed between the interacting LPs.
Based on the above defined precedence relations the *causally precedes* relation is defined as follows:

- $\sigma_i$ *causally precedes* $\sigma_j$ (denoted as $\sigma_i \rightarrow \sigma_j$) if and only if one of the following is true:
  
  a. $(\sigma_i < \sigma_j)$
  
  b. $(\sigma_i \xrightarrow{m} \sigma_j)$
  
  c. $\exists \sigma_k$ such that $\sigma_i \rightarrow \sigma_k$ and $\sigma_k \rightarrow \sigma_j$

The *causally precedes* relation represents a cause-effect relationship between events. Lack of this relationship is called *concurrency*. Events $\sigma_i$ and $\sigma_j$ is said to be *concurrent* (denoted as $\sigma_i \parallel \sigma_j$) if and only if $(\sigma_i \not\leftarrow \sigma_j) \land (\sigma_j \not\leftarrow \sigma_i)$.

Precedence relations between events of a DSS are immediately visible in the general structural model. An execution of a DSS is simply a path in the structural model. If there exists a path from one event (state) to another one, then these two events are causally related. If two events (states) cannot be connected by a path (not accessible in the graph), then these events can occur concurrently in an execution.

We define a message set $Z = \{ m_{ij} : \forall \sigma_i, \sigma_j \in D.S.S_{external} \text{ such that } \sigma_i \xrightarrow{m_j} \sigma_j \}$ for the DSS. Earlier, it was shown in the compatibility of structural models that every output event has a matching input event. The message set assigns a unique identifier to each of these matched external event pairs. Basically, each message corresponds to flow of an entity from the source LP to the destination (or target) LP. Therefore, each message should carry the necessary information regarding the attributes of the entity and its timing information.
3.5 Scalability and Modeling Power of Structural Models

It has been shown in [Mettala, 1989] and [Smith, 1992] that finite state automata based models of discrete event systems grow linearly as the system size grows. This is not a very surprising finding because number of states in an automaton directly represents the number of resources in the system. Therefore, state space explosion is not an issue for the structural models defined in this study, which makes them highly scalable.

Being derived from communicating and timed-state automata (see [Lynch, 1996] for communicating automata and [Cassandras, 1993] for timed-state automata) structural model formalism represents a very versatile modeling tool enabling modeling of the physical structure (part flow) of any discrete event system. Of course, necessary structural mappings must be defined between the specific system components and model components (states and events) for creation of accurate structural models.

It is important to note here that, structural models are designed to model the flow characteristics of discrete event system and not the decision making characteristics. In this respect, structural models are not as powerful as Petri-nets (no mechanisms to model behaviors such as inhibition of flow, synchronized flow, conditional flow etc.). However, for the same reason, structural models are more scalable and less complex compared to Petri-nets.

3.6 The Property of Synchronization in Distributed Simulation Systems

The word ‘synchronization’ has many meanings depending on the context where it is used. In general, synchronization means performance of tasks at the same time by independent parties.

In the context of distributed simulation, synchronization can be described as the execution of independent simulation processes in a coordinated fashion so that causal
relationships between events are protected. In Chapter 1, the situation called a **time ambiguity** was introduced, which is caused by a **straggler event**. A straggler event is an event which is not executed on time and therefore can cause a violation of a causally precedes relationship. This violation is called a time ambiguity because the timing of the simulation process was the main cause for it to happen.

Time ambiguities can also occur locally in a single simulation process if events are not executed in proper order. However, use of an event list where future scheduled events are ordered according to their execution times prevents this. In a distributed simulation setting, since independent simulation processes do not share a single event list, orderly execution of remotely induced events is not guaranteed. If the LVC of a simulation process is ahead of the execution time of a remotely induced event, then the order of execution is immediately violated. This shows that straggler events are the ones that are induced by remote simulation processes, i.e. all input events are potential straggler events.

**Definition 3.1** Let \( \sigma_i \in LP_j \Sigma_{\text{output}} \) and \( \sigma_j \in LP_j \Sigma_{\text{input}} \) be two events such that \( \sigma_i \xrightarrow{m_{ij}} \sigma_j \). If \( LP_j \text{LVC}(t_{\text{receive}}^{m_i}) > LVT(\sigma_j) \), where \( t_{\text{receive}}^{m_i} \) is the wallclock time of receipt of message \( m_{ij} \) at \( LP_j \), then a time ambiguity is said to occur at \( LP_j \) caused by the receipt of the message \( m_{ij} \) from \( LP_i \). Event \( \sigma_j \) is called a straggler event.

The occurrence of time ambiguities is detrimental for distributed simulations, because it causes out of order execution of causally related events which leads to incorrect simulation of the physical process. The existence of time ambiguities render the simulation output questionable and arbitrary.

**Definition 3.2** Synchronization of distributed simulations is the act of coordinating simulation processes so that time ambiguities do not occur or correcting execution of the simulation processes if a time ambiguity occurs.
In the above definition of the synchronization, two possible approaches to deal with time ambiguities are stated:

1. Coordinate simulation processes so that time ambiguities are prevented from happening.
2. Correct the execution of simulation processes when ambiguities occur.

The first approach is a proactive one and known as conservative synchronization and the second approach is a reactive one known as optimistic synchronization. Details of conservative and optimistic synchronization are discussed in Chapter 2.

We provide the following theorem regarding the first approach to synchronization.

**Theorem 3.1** Two simulation processes $LP_k$ and $LP_i$ are said to be synchronized if for all event pairs $(\sigma_i, \sigma_j) \in [LP_k, \Sigma_{\text{external}} \cup LP_i, \Sigma_{\text{external}}]^2$, where $\sigma_i \xrightarrow{m_{ij}} \sigma_j$, $LVT(\sigma_j) \geq LP_x \cdot LVC(t_{\text{receive}}^{m_{ij}})$, where $LP_x$ is the receiver of the message $m_{ij}$.

Theorem 3.1 follows directly from the definition of the time ambiguity. If one can assure that the local virtual clock of the receiver simulation process is always less than the virtual time of execution of input events at the wallclock time of receipt of the corresponding messages then time ambiguities cannot occur and therefore the simulation processes are synchronized.

If all simulation processes of a DSS are synchronized we say that the DSS is synchronized. In this respect, synchronization is a property of a DSS in which time ambiguities are always prevented from happening. As one can understand from the above given theorem, creation and sustaining of the synchronization property in a DSS requires control and coordination of local virtual clocks of the simulation processes in a DSS.
3.7 Analyses of Synchronization Requirements for Paced Execution

In paced execution, the local virtual clock of the simulation process advances in coordination with a real-time clock (i.e., wallclock). This means that, every virtual time unit in the simulation execution corresponds to a real-time duration measured by the wallclock. If the correspondence between the virtual clock and the wallclock is one-to-one, then the simulation is said to advance in real-time. Obviously, for computer simulations used for analysis it is desirable to advance faster than real-time.

Paced execution is not utilized for sequential discrete-event simulation since event-driven execution provides much faster run times. This is simply due to the amount of time wasted while waiting between consecutive virtual time steps in paced execution. However, this mechanism proves very valuable in synchronizing distributed simulations, since synchronization can be established without frequent communication between the processes and also without occurrence of deadlocks.

Using paced execution, all simulation clocks can be advanced at a common pace synchronized with the rate of advance of a reference wallclock. Using this mechanism, it is possible to advance the clocks of the LPs at a fairly synchronous fashion (some error is of course unavoidable) eliminating any possible time ambiguities. Such a mechanism would reduce the amount of communication for coordination tremendously compared to the existing techniques, since the logical processes only communicate to the reference wallclock when their local hardware clocks drift far enough apart to endanger synchronicity.

System clocks present in today’s personal computers are simple electronic clocks that use Quartz crystals as oscillators for determining the clock frequency. Unlike very expensive Cesium or Rubidium crystals used in atomic clocks, Quartz crystals are not very precise oscillators and their frequencies are sensitive to environmental conditions, especially to temperature changes. Therefore, a regular system clock embedded in a
desktop computer’s motherboard would have significant drifts (ranging from couple seconds to couple minutes a day depending on temperature changes) during operation. For this reason, it is necessary to synchronize system clocks of computers running logical processes if paced simulation is to be used for synchronizing the DSS.

Current technology provides low-cost tools such as Global Positioning System (GPS) receivers which can synchronize the system clock of a personal computer to the clock of a GPS satellite by a maximum error of 1 microsecond in a very fast way without using any network bandwidth. GPS satellites use atomic clocks to determine time and broadcast timing information all over the world. A GPS receiver connected to each computer in a DSS can efficiently synchronize system clocks without using any bandwidth from the communication network. Availability of low cost GPS receivers makes this technique a practical solution to the system clock synchronization issue.

On the other hand, Network Time Protocol (NTP) also offers a reliable way for synchronizing system clocks. However, unlike the previous method, NTP requires a communication network for synchronization, but the bandwidth requirements are optimized. Using NTP, system clocks within a sub-network can be synchronized to the accuracy of 1 millisecond using low cost networking technology. Higher accuracies are attainable by using faster network equipment. A detailed explanation of NTP and the related clock synchronization algorithms can be found in [Mills, 1994].

The above discussion clearly shows that one can very accurately synchronize hardware clocks of all LPs and use these hardware clocks to pace the advance of the time steps during execution resulting in synchronous time-stepped execution for the entire DSS. Let’s examine how coordinated time-stepped execution can provide a solution to the synchronization problem.
3.7.1 Ideal Case – Perfect Clock Synchronization

Let $D$ be a DSS defined by the 2-tuple $<D_{LP}, T_D>$ where $D_{LP} = \{LP_1, LP_2, ..., LP_i, ..., LP_N\}$ and $T_D$ is a directed graph defining the topology of $D$. Logical processes are interconnected by communication links defined by the connection topology $T_D$. $T_D$ is represented as an adjacency matrix (or a directed graph) as follows:

$$T_D = [T_{D_{ij}}] \text{ such that } T_{D_{ij}} = \begin{cases} 1 \text{ iff } LP_i \text{ can send a message to } LP_j \\ 0 \text{ otherwise} \end{cases}$$

(3.4)

$$1 \leq i, j \leq N \text{ where } i, j, N \text{ are integers}$$

Note that $[T_{D_{ij}}]$ is a size $N$ square matrix composed of binary values indicating the directed communication links among the LPs. Let $LVT_i(t)$ denote $LVT(LP_i)$ at time $t$. Also, let $e_i^k$ be the $k^{th}$ event executed on $LP_i$ during the execution of $D$ and $m_{ij}^k$ be the $k^{th}$ message sent from $LP_i$ to $LP_j$. Each message is associated with a communication delay denoted by $\Delta cm_{ij}^k$, a simulated transportation time denoted by $\Delta Tm_{ij}^k$ and a sending timestamp (equal to $LVT_i$ at the time of sending) denoted by $TSm_{ij}^k$.

Assume that the execution of all LPs start at exactly the same time (this assumption will be relaxed in the next section). At each time step, each LP will spend exactly the same amount of wallclock time and therefore all LVTs will advance at the same rate. Since all LPs start at the same time with all LVTs set to zero initially, at any time during an execution all LVTs should be the same assuming no hardware clock drift (perfectly synchronized hardware clocks). Thus:

$$\text{If } LVT_i(t_0) = LVT_j(t_0) = 0 \text{ and } \frac{dLVT_i}{dt} = \frac{dLVT_j}{dt} \forall LP_i, LP_j \in D_{LP}$$

then $LVT_j(t_k) = LVT_i(t_k) \forall t_k > 0 \text{ and } LP_i, LP_j \in D_{LP}$

(3.5)
\( \frac{dLVT_i}{dt} \) is the rate of advance of the simulation clock of \( LP_i \), with respect to the wallclock. For convenience, the following definition is introduced in which \( k \) denotes the “time scaling factor”:

\[
\frac{dLVT_i}{dt} = \frac{1}{k} \forall \ LP_i \in D_{LP} \tag{3.6}
\]

Notice that \( k = 1 \) means that the simulation is running at wallclock speed, i.e. real-time simulation. As the value of \( k \) decreases (\( k < 1 \)) simulation speed increases.

Assume that at wallclock time \( t_i \), event \( e_i^k \) at \( LP_i \) sends the message \( m_{ij}^1 \) to \( LP_j \) with a timestamp \( TSm_{ij}^1 \) that is equal to \( LVT_i(t_i) \). Message \( m_{ij}^1 \) has a communication delay of \( \Delta cm_{ij}^1 \) and an associated simulated non-zero transportation time of \( \Delta Tm_{ij}^1 \). This message will try to schedule an event \( e_j^k \) with a timestamp equal to \( TSm_{ij}^1 + \Delta Tm_{ij}^1 \). At the time of the receipt of this message simulation clock of \( LP_j \) will show the value \( LVT_j(t_i) + \frac{dLVT_j}{dt} \cdot \Delta cm_{ij}^1 \). In order not to have a time ambiguity at \( LP_j \), when event \( e_j^k \) is scheduled by the message \( m_{ij}^1 \) from \( LP_i \), the following condition should be satisfied:

\[
LVT_j(t_i) + \frac{dLVT_j}{dt} \cdot \Delta cm_{ij}^1 < TSm_{ij}^1 + \Delta Tm_{ij}^1 \tag{3.7}
\]

Equivalently (since \( TSm_{ij}^1 = LVT_i(t_i) \)) one can write:

\[
LVT_j(t_i) + \frac{dLVT_j}{dt} \cdot \Delta cm_{ij}^1 < LVT_j(t_i) + \Delta Tm_{ij}^1 \tag{3.8}
\]

And this simplifies to:

\[
\frac{dLVT_j}{dt} \cdot \Delta cm_{ij}^1 < \Delta Tm_{ij}^1 \tag{3.9}
\]

Due to Eq. 3.6 one can write:

\[
\frac{1}{k} \Delta cm_{ij}^1 < \Delta Tm_{ij}^1 \tag{3.10}
\]
This is equivalent to (assuming that $\Delta Tm_{ij}^1$ is not zero):

$$k > \frac{\Delta cm_{ij}^1}{\Delta Tm_{ij}^1}$$ \hspace{1cm} (3.11)

This inequality clearly shows a lower bound on the time scaling factor (or an upper bound on the rate of advance of the simulation clock) determined by the parameters of a single message passed between two LPs. The lower bound is directly proportional to the communication delay and it is inversely proportional to the simulated transportation time for the entity being transferred between the simulation processes.

In general, in a distributed simulation system $D$, every communication link and message type will limit the scaling factor. The common scaling factor for $D$ will be the maximum of these lower bounds dictated by all of the messages being passed in the system. Let $k_D$ be the maximum of these lower bounds, then:

$$k_D > \max \{k_i : \forall i, LP_i \in D\} \text{ where }$$

$$k_i = \max \left\{ \frac{\Delta cm_{ij}^k}{\Delta Tm_{ij}^k} : \forall k, \forall j, LP_j \in D, j \neq i, T_{D_i} = 1 \right\}$$ \hspace{1cm} (3.12)

Here $k_i$ denotes the maximum of lower bounds calculated for the messages of $LP_i$ and $k_D$ is the overall maximum of lower bounds for all LPs in $D$.

The scaling factor $k_D$ is the maximum amount one can accelerate the advance of the simulation clock with respect to the wallclock time. It is important to realize that the overall lower bound on the scaling factor depends on the maximum ratio of the communication delay to the simulated transportation time for all messages being passed during the execution of a system. In order to maximize speed of the execution, $k_D$ should be as small as possible. The communication delay in the system should be minimized and the amount of simulated transportation time in the simulation models should be maximized. However, the latter parameter is not under the control of the execution system since it is a property of the physical system being simulated. Notice that if the
transportation time is zero (or similarly the communication delay is infinity) for a single message in the system, the lower bound on the scaling factor will be infinity implying a not advancing execution. This means that, if two consecutive communication events (events that send out messages and affect other LPs) have the same timestamp (or maximum communication time is unbounded for a message), then the DSS should suspend the advance of the LVTs until these events are processed.

**3.7.2 Realistic Case – Imperfect Clock Synchronization**

It is not possible in practice to synchronize system clocks of networked computers perfectly. Due to communication latency and error-prone clock oscillators, regardless of the synchronization technique used, there is always some error involved in synchronizing system clocks of computers. This being the case, the error-free model introduced in the previous section cannot be successfully implemented without adding further analysis considering the clock synchronization and other errors.

In this respect, one can define two major error sources inherently present in a paced execution mechanism:

1. **System Clock Synchronization Error**: This is the maximum offset between the reference clock and the system clock being synchronized at the time of synchronization and defined as follows:

   \[ |SC_i(t_s) - t_s| \leq \epsilon_{\text{max}} \]

   Where \( SC_i(t_s) \) represents the value of the system clock of \( LP_i \) at time \( t_s \) when the synchronization takes place and \( \epsilon_{\text{max}} \) denotes the maximum synchronization error.

2. **System Clock Frequency Drift**: This is the drift of system clock frequency from the true reference clock frequency that occurs after synchronization. The frequency of the system clock \( SC_i \) is denoted by:
\[ f_i(t) = \rho_i^{-1}(t) = \left[ \frac{dSC_i(t)}{dt} \right]^{-1} \]  
\text{(3.14)}

Where \( \rho_i(t) \) is the rate of advance of \( SC_i \). At the time of synchronization, the frequency of the system clock being synchronized is the same as that of the reference clock, that is:

\[ \rho_i(t_s) = \rho_{ref} \]  
\text{(3.15)}

There is a rate of drift for every system clock frequency which is determined by environmental factors (i.e. random) but has an absolute maximum due to the nature of the oscillator:

\[ \frac{d\rho_i(t)}{dt} = \rho'_i(t) \text{ and } |\rho'_i(t)| \leq \rho'_\text{max} \]  
\text{(3.16)}

In the presence of the errors type 1 and 2, obviously the assumption that all LPs start paced execution at the exact same time cannot be achieved. So, once this assumption is relaxed a realistic analysis of the limit on the speed factor can be achieved.

Let \( LP_i \) and \( LP_j \) agree to start paced execution at time \( t_i \) and also assume that \( SC_i \) and \( SC_j \) were last synchronized to the reference clock at time \( t_0 < t_i \). Assume that clocks are periodically synchronized with the reference clock \( SC_{ref} \) every \( t_{sync} \) time units.

At time \( t_0 \), it is known (due to synchronization error) that:

\[ |SC_i(t_0) - t_0| \leq \epsilon_{\text{max}} \text{ and } |SC_j(t_0) - t_0| \leq \epsilon_{\text{sync}} \]  
\text{(3.17)}

Therefore, it is also known that the maximum gap between \( SC_i \) and \( SC_j \) at time \( t_0 \) is:

\[ |SC_i(t_0) - SC_j(t_0)| \leq 2\epsilon_{\text{sync}} \]  
\text{(3.18)}
Assume the worst case such that:

\[ SC_i(t_0) - SC_j(t_0) = 2e^{\text{sync}}_{\text{max}} \]  \hspace{1cm} (3.19)

This implies that \( SC_i \) will reach \( t_i \) before \( SC_j \) and there will be an offset between the starting times of these two LPs. Assume that the LP with the slower clock (i.e. \( LP_j \)) starts at time \( t_{k+1} \) and the LP with the faster clock (i.e. \( LP_i \)) starts paced execution at time \( t_k \). This yields:

\[ SC_i(t_k) = t_i \quad \text{and} \quad SC_j(t_{k+1}) = t_i \quad \text{where} \quad t_{k+1} > t_k \]  \hspace{1cm} (3.20)

The objective is to calculate the maximum possible wallclock time difference between the starting times of \( LP_i \) and \( LP_j \) (i.e. \( \max(t_{k+1} - t_k) \)). The maximum gap occurs when \( SC_i \) and \( SC_j \) drift at the maximum rate in opposite directions:

\[ \rho'_i(t) = \rho'_{\text{max}} \quad \text{and} \quad \rho'_j(t) = -\rho'_{\text{max}} \quad \forall t \geq t_0 \]  \hspace{1cm} (3.21)

This situation is depicted in Figure 3-7.

---

**Figure 3-7:** Worst case behavior of system clocks during execution

At time \( t_0 \) (i.e. time of synchronization), both clocks have the same clock frequency as the reference clock:
\[ \rho_j(t_0) = \rho_j(t_0) = \rho_{ref} \] (3.22)

Using this initial value information we can derive the clock rate function as follows:

\[
\rho_j(t) = -\rho'_{\text{max}} t + c \Rightarrow c = \rho_{ref} + \rho'_{\text{max}} t_0
\] (3.23)

At time \( t_{k+1} \), the value of \( SC_j \) is given by:

\[
SC_j(t_{k+1}) = SC_j(t_0) + \int_{t_0}^{t_{k+1}} \rho_j(t) dt
\] (3.24)

Substituting the clock rate function yields:

\[
SC_j(t_{k+1}) = SC_j(t_0) + \int_{t_0}^{t_{k+1}} (-\rho'_{\text{max}} t + \rho_{ref} + \rho'_{\text{max}} t_0) dt
\]
\[ = SC_j(t_0) + \left[ -\rho'_{\text{max}} \frac{t^2}{2} + \rho_{ref} t + \rho'_{\text{max}} t_0 t \right]_{t_0}^{t_{k+1}} \]
\[ = SC_j(t_0) + \left[ -\rho'_{\text{max}} \frac{(t_{k+1}^2 - t_0^2)}{2} + \rho_{ref} (t_{k+1} - t_0) + \rho'_{\text{max}} (t_{k+1}^2 - t_0^2) \right] \]
\[ = SC_j(t_0) + [(t_{k+1} - t_0)(\rho_{ref} + \rho'_{\text{max}} t_0 - \frac{\rho'_{\text{max}}}{2} (t_{k+1} + t_0))] \]
\[ = SC_j(t_0) + (t_{k+1} - t_0)[(\rho_{ref} + \frac{\rho'_{\text{max}}}{2} t_0 - \frac{\rho'_{\text{max}}}{2} t_{k+1})] \]
\[ = SC_j(t_0) + (t_{k+1} - t_0)[(\rho_{ref} - \frac{\rho'_{\text{max}}}{2} (t_{k+1} - t_0))] \]

On the other hand, the clock rate function for \( SC_i \) is:

\[ \rho_i(t) = \rho'_{\text{max}} t + \rho_{ref} - \rho'_{\text{max}} t_0 \] (3.26)
Using a similar approach, the value of $SC_i$ at time $t_k$ is found as:

$$SC_i(t_k) = SC_i(t_0) + \int_{t_0}^{t_k} (\rho'_{\text{max}} t + \rho'_{\text{ref}} - \rho'_{\text{max}} t_0) dt$$

$$= SC_i(t_0) + \left[ \rho'_{\text{max}} \frac{t_k^2}{2} + (\rho'_{\text{ref}} - \rho'_{\text{max}} t_0) t \right]_0^t$$

$$= SC_i(t_0) + \left[ \rho'_{\text{max}} \frac{(t_k - t_0)^2}{2} + (\rho'_{\text{ref}} - \rho'_{\text{max}} t_0)(t_k - t_0) \right]$$

$$= SC_i(t_0) + (t_k - t_0) \left[ \rho'_{\text{ref}} + \frac{\rho'_{\text{max}}}{2} (t_k - t_0) \right]$$

(3.27)

Notice that $SC_j(t_{k+1})$ and $SC_i(t_k)$ are very similar except for the sign difference of the term $\frac{\rho'_{\text{max}}}{2}(t - t_0)$. This is obviously due to the clock drift rate difference between the two clocks.

Since $SC_i(t_k) = SC_j(t_{k+1}) = t_1$:

$$SC_j(t_0) + (t_{k+1} - t_0) \left[ \rho'_{\text{ref}} - \frac{\rho'_{\text{max}}}{2} (t_{k+1} - t_0) \right] =$$

$$SC_i(t_0) + (t_k - t_0) \left[ \rho'_{\text{ref}} + \frac{\rho'_{\text{max}}}{2} (t_k - t_0) \right]$$

(3.28)
Using the synchronization error at time $t_0$ (Eq. 3.20) the following is obtained:

\[
(t_{k+1} - t_0) \left[ \rho_{ref} - \frac{\rho_{max}}{2} (t_{k+1} - t_0) \right] - (t_k - t_0) \left[ \rho_{ref} + \frac{\rho_{max}}{2} (t_k - t_0) \right] = 2e_{sync}^\text{max}
\]

\[
\Rightarrow \rho_{ref} (t_{k+1} - t_k) - \frac{\rho_{max}}{2} \left[ (t_{k+1} - t_0)^2 + (t_k - t_0)^2 \right] = 2e_{sync}^\text{max}
\]

\[
\Rightarrow t_{k+1} - t_k = \frac{2e_{sync}^\text{max} + \rho_{max}^\prime t_{sync}^2}{\rho_{ref}}
\]

(3.29)

Again, considering the worst case that yields the maximum gap between the starting times of $LP_i$ and $LP_j$, we have $t_{k+1} - t_0 \leq t_{sync}$ and $t_k - t_0 < t_{sync}$, an upper bound on the gap is given by:

\[
t_{k+1} - t_k < \frac{2e_{sync}^\text{max} + \rho_{max}^\prime t_{sync}^2}{\rho_{ref}} \leq t_{\text{gap}}^\text{max}
\]

(3.30)

Eq. 3.31 defines an upper bound on the maximum gap starting paced execution between any two logical processes within a DSS. If this gap occurs between the sender and receiver logical processes, it is possible to observe the worst case situation such that the faster logical process ($LP_i$) is the receiver. In this case, the receiver’s LVT ($LP_i, LVT$) would advance beyond the sender’s LVT ($LP_j, LVT$) during paced execution by the amount ($k$ denotes the scaling factor used during the paced execution period):

\[
\frac{t_{\text{gap}}^\text{max}}{k} \leq T_{\text{max}}^\text{gap}
\]

(3.31)

This indicates that, it is possible to observe a time ambiguity if the scaling factor is calculated without accounting for this gap. Following this result, in a DSS with imperfect system clock synchronization one can state that for any message $m_{ji}$ exchanged between any two logical processes $LP_j$ and $LP_i$, the lower bound on the execution time scaling factor is given by:
\[ k > \frac{\Delta cm_{ji}}{\Delta Tm_{ji} - T_{\text{gap}}^\text{max}} \]  

(3.32)

Substituting \( T_{\text{gap}}^\text{max} \) with its expanded form we get:

\[ k > \frac{\Delta cm_{ji}}{2\varepsilon_{\text{sync}}^{\text{max}} + \rho_{\text{max}}' t_{\text{sync}}^2} \]  

(3.33)

Solving for the time scaling factor yields the result:

\[ k > \frac{\Delta cm_{ji} + \frac{2\varepsilon_{\text{sync}}^{\text{max}} + \rho_{\text{max}}' t_{\text{sync}}^2}{\rho_{\text{ref}}}}{\Delta Tm_{ji}} \]  

(3.34)

This lower bound on the scaling factor can be further generalized by substituting a maximum communication delay (\( \Delta c_{\text{max}} \geq \Delta cm_{ji}, \forall m_{ij} \)) for the communication time for any messaging in the system and we get the general upper bound:

\[ k > \frac{\Delta c_{\text{max}} + \frac{2\varepsilon_{\text{sync}}^{\text{max}} + \rho_{\text{max}}' t_{\text{sync}}^2}{\rho_{\text{ref}}}}{\Delta Tm_{ji}} \]  

(3.35)

This result is quite intuitive since one would expect any inefficiency in the synchronization and communication to add up to increase the lower bound on the time scaling factor and hence result in slower execution speed.

One important observation following this result is that, the synchronization time interval, \( t_{\text{sync}} \), proves to be an important factor along with the communication delay (all other parameters \( \varepsilon_{\text{sync}}^{\text{max}}, \rho_{\text{ref}} \) and \( \rho_{\text{max}}' \) are limited by the technology/hardware used) that should be carefully adjusted for top performance. Clearly, this is due to the tradeoff that exists between the frequency of hardware clock synchronization and the messaging overhead it creates (extra communication bandwidth consumed by hardware clock synchronization messages).
Chapter 4

Automated Synthesis of Structural Models

Structural models play a central role in the building of local synchronization agents (LSAs). The general structural model for a logical process is embedded in the LSA as well as entity specific structural models. Prior to execution, a preprocessor analyzes the structural models and generates state-specific look-up tables that hold timing information for predicting minimum virtual time to all possible output events. During execution, an LSA uses structural models to keep track of entity states and the look-up tables to predict the minimum time to reach possible output events.

Structural models are the key components of the modular synchronization framework developed in this study. In fact, all components of a local synchronization agent other than the structural model core are standard (see Chapter 5 for components of the local synchronization agents). It is the structural model core that customizes an LSA and enables it to connect with a specific logical process (i.e., a simulation model).

Therefore, generation of an LSA starts with the synthesis of the structural models from the corresponding logical process. The synthesis of structural models is the most intricate part of creating a distributed simulation system within this framework.

Certain rules must be established for the mapping of simulation elements in logical processes to states and events in structural models. Since different commercial-off-the-shelf (COTS) simulation packages use different simulation modeling terms, notations, and languages, mapping rules would be different for different COTS simulation packages.
In this research, the Arena™ simulation package (version 7.01) from Rockwell Software was used to demonstrate the synthesis of structural models. Interoperability of different COTS simulation packages is currently a research area by itself, and is not in the interest of this research. Therefore, only a single COTS simulation package was used within the DSS implementation. For this reason, the software tools developed for the implementation of the findings of this study are specifically designed for the Arena™ simulation package. However, the findings of this study can be applied to many of the currently available COTS simulation packages. The concept of modular structural models for synchronization can be further developed to encompass a variety of COTS applications to remedy the interoperability problem.

Creation of a rule-base for mapping logical process elements to structural model elements is a one time operation that is done for every different simulation modeling language. When the rule-base is combined with necessary mechanisms for reading and parsing the models from the target simulation modeling language, a complete method for generating structural models would be in hand.

One of the objectives of this research is to automate the synthesis of the synchronization mechanism. Within the modular framework developed here, this goal translates into automated synthesis of the structural models. For this purpose, algorithms are developed and implemented to automate the parsing and mapping of simulation processes to required structural models.

The chapter is organized as follows. First, the assumptions for creation of Arena™ models are stated and the domain of implementation is discussed in Section 1. Next, in Section 2, an example demonstrating mapping of an Arena™ simulation model to structural models is provided. Last, in Section 3, an overview of the synchronization architecture in relation to structural models is provided.
4.1 Implementation Domain and Modeling Assumptions

Arena™ simulation package is commercial grade simulation software built on a very comprehensive simulation modeling platform that has numerous commands and constructs useful for modeling virtually any type of manufacturing and service operation.

The synthesis of structural models while using a specific modeling platform requires mapping of platform specific constructs to structural model components. A realistic and commercial grade implementation would require the creation of a complete mapping that covers all constructs in the target modeling platform. However, a complete mapping would require a substantial amount of analysis and coding effort.

The purpose of this implementation is to show the feasibility and demonstrate the advantages of using structural models to automatically generate custom code for distributed synchronization agents. Therefore, a limited implementation domain (which can be modeled using a small number of modeling constructs) is selected for demonstration to illustrate the suitability of this concept, yet complicated enough to test different aspects of the modeling formalism. A complete mapping of the target simulation platform is beyond the scope of this research.

Models of queuing networks form the domain for implementation. Therefore, only a subset of the constructs of the Arena™ simulation platform is selected for the structural mapping. The selection was made so that the collection of selected constructs is adequate to model reasonably complicated queuing networks.

In Section 4 of Chapter 3, the structural model formalism with the necessary modeling power to represent the flow logic of entities in any discrete event system was presented. This domain-limited implementation does not imply any loss of generality regarding modeling capabilities and scalability of structural models.
The Arena™ simulation package provides an easy to learn and user-friendly graphical modeling environment for creating simulation processes of physical or imaginary systems. A brief explanation of the Arena™ modeling environment along with the modeling constructs selected to be used in this implementation is provided in Appendix A. A detailed and complete explanation of the Arena™ modeling language syntax and general information on developing models with Arena™ can be found in [Kelton et al., 2003].

Appendix B provides a detailed explanation of the mapping of the selected Arena™ elements to structural modeling components. Arena™ elements are mapped into small modular structural model components, which are formed by states and events. These components are then assembled together based on the sequence of Arena™ elements in the simulation model to form the structural model.

Appendix C provides the data structures used to implement structural model components.

### 4.2 A Sample Structural Model

In this section, an example that demonstrates mapping of Arena™ simulation models to structural model components is presented. The example physical process (PP₁) is composed of six servers. Each server has an infinite capacity (adequately large) input buffer. There are two types of parts being processed in the system, named A and B.

Type A parts originate locally in the system (e.g., raw material) where as type B parts are received from another physical process (PP₂) and further processed in the system (e.g., preprocessed part). There is also a third physical process (PP₃) to which some percentage of the finished parts are transported. The physical process PP₁ is depicted in Figure 4-1.
The flow of parts through the servers is represented by arrows. Each arrow has a label that denotes the part type(s) allowed for that transfer and the probability of a part making the transfer. For example, the green arrow marked with $B$, 1 from server 3 to server 5 means only parts of type $B$ can do this transfer and the probability of the transfer taking place is 1.

Upon completion of processing, parts either leave for another physical process ($PP_2$ or $PP_3$) or they are terminated at server 6 (scrap parts) with a probability of 0.25.

The interarrival time for type A parts is uniformly distributed between 10 and 20 minutes. The interarrival time for type B parts is unknown since parts arrive from another process. However, the transportation time for any part between any two physical processes is 2 minutes. Processing times of the parts at different servers are given in Table 4-1.
Table 4-1: Processing time of the parts in the example system (minutes)

<table>
<thead>
<tr>
<th></th>
<th>Part Type A</th>
<th>Part Type B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server 1</td>
<td>Triangular (5, 10, 15)</td>
<td>Uniform (13, 18)</td>
</tr>
<tr>
<td>Server 2</td>
<td>Uniform (3, 8)</td>
<td>Triangular (12, 16, 20)</td>
</tr>
<tr>
<td>Server 3</td>
<td>Uniform (1, 4)</td>
<td>Triangular (8, 10, 12)</td>
</tr>
<tr>
<td>Server 4</td>
<td>Triangular (2, 4, 6)</td>
<td>x</td>
</tr>
<tr>
<td>Server 5</td>
<td>x</td>
<td>Triangular (6, 10, 14)</td>
</tr>
<tr>
<td>Server 6</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

The corresponding Arena™ simulation model (LP₁) for this physical system (PP₁) is given in Figure 4-2. This is a graphical representation of the model and does not provide information on the module parameters. Module parameters and other details are omitted in this section in order to provide a concise example of structural model mapping.

The general structural model for this Arena™ simulation model of the sample physical process PP₁ is provided in Figure 4-3. Formal definition of this general structural model is given in Table 4-2. Entity types are mapped as $\varepsilon_A$ for parts of type A and $\varepsilon_B$ for parts of type B.

In the structural model, the set of enabled events ($\Gamma$) is formed by examining the decision modules that have conditions on the entity type. In this specific example, the decision module named Decide2 is the only module that has a condition on the entity type attribute. All other decision modules have probabilistic conditions and therefore they are replaced with direct transitions during mapping.
Figure 4-2: Arena™ simulation model – graphical representation
Table 4-2: Formal definition of the general structural model

\[ L_{P_1,S} = (X, \Sigma, \Gamma, \Psi, B, \overline{X}) \text{ and } E = \{ e_A, e_B \} \]

\[ X = \{ X_0, X_1, \ldots, X_{11}, X_{12} \}, \overline{X} = \{ X_0, X_2 \} \]

\[ \Sigma = \{ \sigma^{(0,1)}_{\text{enter}}, \sigma^{(2,3)}_{\text{create}}, \sigma^{(1,4)}_{\text{process}}, \sigma^{(1,5)}_{\text{process}}, \sigma^{(1,6)}_{\text{process}}, \sigma^{(3,4)}_{\text{process}}, \sigma^{(3,5)}_{\text{process}}, \sigma^{(3,6)}_{\text{process}}, \sigma^{(4,7)}_{\text{process}}, \sigma^{(4,8)}_{\text{process}}, \sigma^{(5,7)}_{\text{process}}, \sigma^{(5,8)}_{\text{process}} \} \]

\[ \sigma^{(6,7)}_{\text{process}}, \sigma^{(7,10)}_{\text{transport}}, \sigma^{(7,9)}_{\text{process}}, \sigma^{(8,9)}_{\text{process}}, \sigma^{(8,12)}_{\text{transport}}, \sigma^{(9,11)}_{\text{dispose}} \}

\[ \Sigma_{\text{Input}} = \{ \sigma^{(0,1)}_{\text{enter}} \}, \Sigma_{\text{Output}} = \{ \sigma^{(7,10)}_{\text{transport}}, \sigma^{(8,12)}_{\text{transport}} \} \]

\[ \Gamma(X_0, e_A) = \emptyset, \Gamma(X_0, e_B) = \sigma^{(0,1)}_{\text{enter}}, \Gamma(X_1, e_A) = \emptyset, \Gamma(X_1, e_B) = \emptyset \]

\[ \Gamma(X_2, e_A) = \sigma^{(2,3)}_{\text{create}}, \Gamma(X_2, e_B) = \emptyset, \Gamma(X_2, e_B) = \emptyset \]

\[ \Gamma(X_3, e_A) = \sigma^{(3,4)}_{\text{process}}, \Gamma(X_3, e_A) = \sigma^{(3,5)}_{\text{process}}, \Gamma(X_3, e_A) = \sigma^{(3,6)}_{\text{process}} \]

\[ \Gamma(X_4, e_A) = \sigma^{(4,7)}_{\text{process}}, \Gamma(X_4, e_B) = \sigma^{(4,8)}_{\text{process}}, \Gamma(X_5, e_A) = \sigma^{(5,7)}_{\text{process}}, \Gamma(X_5, e_B) = \sigma^{(5,8)}_{\text{process}} \]

\[ \Gamma(X_6, e_A) = \sigma^{(6,7)}_{\text{process}}, \Gamma(X_6, e_B) = \sigma^{(6,8)}_{\text{process}}, \Gamma(X_7, e_A) = \Gamma(X_7, e_B) = \sigma^{(7,10)}_{\text{transport}}, \sigma^{(7,9)}_{\text{process}} \]

\[ \Gamma(X_8, e_A) = \Gamma(X_8, e_B) = \sigma^{(8,9)}_{\text{process}}, \sigma^{(8,12)}_{\text{transport}}, \Gamma(X_9, e_A) = \Gamma(X_9, e_B) = \sigma^{(9,11)}_{\text{dispose}} \]

\[ \Gamma(X_{10}, e_A) = \Gamma(X_{10}, e_B) = \emptyset, \Gamma(X_{11}, e_A) = \Gamma(X_{11}, e_B) = \emptyset, \Gamma(X_{12}, e_A) = \Gamma(X_{12}, e_B) = \emptyset \]

\[ \Psi : \text{Represented by the graph in Figure 4-3} \]

\[ B(l_X) = \begin{bmatrix} 0 & 0 & 10 & 0 & 5 & 3 & 1 & 2 & 6 & 3 & 0 & 0 & 0 \end{bmatrix} \]

\[ B(u_X) = \begin{bmatrix} 2 & 0 & 20 & 0 & 18 & 20 & 12 & 6 & 14 & 3 & 0 & 0 & 0 \end{bmatrix} \]
The bound-map \( B \) is formed by determining the minimum and maximum possible delay period at each state for all entity types (since this is the general model). The first vector \( B(l) \) has the lower bound values for all ordered from left to right in increasing index values. Similarly, the second vector \( B(u) \) lists the upper bound values for all states.

For example, for state \( \chi_5 \) (represents processing in server 2), the minimum processing time is 3 minutes (part type A) and the maximum processing time is 20 minutes (part type B).

States \( \chi_1, \chi_3, \chi_{10}, \chi_{11}, \chi_{12} \) have zero lower and upper bound values, because entities do not actually spend any time in these states (these are either states coming right after entrance to the model or final states).
Calculation of the time bounds for internal and final states is straight-forward and done by examining the process modules minimum and maximum delay values. For initial states, we need to consider the interarrival time (for the CREATE module) or the transportation time (for the ENTER module).

In the mapping of a CREATE module, the bound values of the corresponding initial state is given by the minimum and maximum values of the interarrival time. On the other hand, an ENTER module cannot directly extract interarrival times from the simulation model because they depend on the departure process of the logical process where entities arrive from.

Consequently, bounds for the initial state $\chi_0$ have a special case. This is the state where entities of type $B_e$ arrive from LP$_2$ (the logical process for PP$_2$). The time an entity arriving from an external process spends in the initial state is given by the residual virtual clock value at its arrival.

The residual virtual time is the virtual time difference between the value of the local virtual clock at the time of the receipt of the corresponding message ($m_{enter}^{(0,1)}$ in this case) and the scheduled virtual time of arrival for the arriving entity. Eq. 4.1 gives the formula for the residual virtual time of an arriving entity.

$$RT_{e_k} = (LP_i,LVT(t_{send}) + \Delta T) - LP_j,LVT(t_{receive})$$

Here, $RT_{e_k}$ denotes the residual virtual time for entity $e_k$, $t_{send}$ is the time (real-time) at which the message is sent from the source logical process ($LP_i$), $t_{receive}$ is the time at which the message is received at the target logical process ($LP_j$) and $\Delta T$ is the transportation time.
For any logical process the minimum value of the residual virtual time is always zero, because there is nothing that prevents a message from arriving at the same time as the local virtual clock of the target logical process is equal to the scheduled time of arrival. Of course, residual virtual time cannot be negative provided that the DSS is synchronized.

On the other hand, if instantaneous message transmission and perfectly synchronized simulation clocks are assumed, then the maximum residual virtual time would be equal to the maximum value of the transportation time. Consequently, the bounds for the initial state $\chi_0$ in our example are 0 and 2 minutes.

The general structural model given in Figure 4-3 is an aggregation of two entity-specific structural models. Once the set of enabled events is determined, one can create the entity-specific structural models for each entity type. Figure 4-4 and Figure 4-5 provide these entity-specific structural models for entity types $e_A$ and $e_B$, respectively.
All details of the formal definitions for the entity-specific models will not be specifically discussed, since most of them are obvious from the graphical representation and the general structural model formal definition.

There are two main differences between the formal definitions of the general and entity-specific structural models. The first one is that entity specific models do not have a set of enabled events (see Chapter 3 for further explanation). The second difference is the bound-maps.

The bound-map for the entity-specific structural model is as follows (elements of the vectors are in the order of increasing index of the defined states):

\[
LP_{1}S_{\epsilon_{\chi}}B(l_{\chi}) = [10 \ 0 \ 5 \ 3 \ 1 \ 2 \ 3]
\]
\[
LP_{1}S_{\epsilon_{\chi}}B(u_{\chi}) = [20 \ 0 \ 15 \ 8 \ 4 \ 6 \ 3]
\]
\[
LP_{1}S_{\epsilon_{\chi}}B(l_{\chi}) = [0 \ 0 \ 3 \ 12 \ 8 \ 6 \ 3]
\]
\[
LP_{1}S_{\epsilon_{\chi}}B(u_{\chi}) = [2 \ 0 \ 18 \ 20 \ 12 \ 14 \ 3]
\]

Up to this point, the definition of the structural model formalism and their synthesis are presented. Structural models form the core of the synchronization
architecture developed in this research. In the next section, a brief overview of the synchronization architecture is provided along with the specific role and place of structural models within this architecture as a prelude to the more detailed discussion of the overall synchronization framework in Chapter 5.

4.3 Synchronization Architecture and Structural Models

In the synchronization architecture, all logical processes are coupled with a Local Synchronization Agent (LSA), which is automatically generated from the simulation model. Each LSA has direct control over the speed of its coupled logical process. LSAs predict output messages of logical processes using state automata based structural models and share this information among themselves. Thus, every LSA has a global picture of all potential messages. LSAs use this information to calculate the necessary speed limit for each potential message and change the speed accordingly at the earliest possible time of message transmission.

In the heart of an LSA is the structural model representation of its coupled simulation process. LSAs use structural models to keep track of entity states in the logical processes. Specifically, entity-specific structural models are used to predict potential output messages from the local logical process. Local synchronization agents use entity-specific structural models to track the entity states and periodically calculate minimum virtual time to start of transmission of outbound messages.

A local synchronization agent is composed of two main components:

- Speed control engine (SCE)
- Interaction management engine (IME)
Speed control engine calculates the speed-limit for the upcoming interaction in the distributed simulation system and re-adjusts the real-time scaling factor of the coupled simulation process until the interaction successfully takes place. Once the interaction is over, SCE goes back to the nominal simulation speed agreed in the DSS. SCE takes direct input from the message prediction and management engine.

Message prediction and management engine interacts with the coupled simulation process by reading the state of simulation entities periodically and updating the state of embedded entity-specific structural model instances. IME creates an instance of the entity-specific structural models for each active entity in the simulation process and terminates instances when entities are disposed from the simulation process.

IMEs of LSAs communicate among each other via special messages called “s-messages” (short for synchronization messages - to distinguish them from regular messages sent/received by simulation processes). These s-messages inform other LSAs about the potential messages and their timing. Each IME uses three lists to manage information about local and system-wide potential messages. These are:

1. Predicted Output Messages (POM) list
2. Predicted Interaction Messages (PIM) list.
3. Active Interaction Messages (AIM) List

POM is a locally-populated list which contains the potential output messages to be sent by the local simulation process. An LSA periodically broadcasts new information on the POM list to other LSAs using s-messages.

PIM is a remotely and locally populated list, which contains potential interaction messages for all LPs in the DSS. SCE directly accesses the PIM list for timely adjustment of the simulation process speed. When IME sends or receives an s-message, it copies the related information (unique entity identification and predicted minimum time to
transmission) to the PIM list. Therefore, every PIM list in the DSS contains the same information which tells what messages are potentially going to be transmitted and the earliest virtual time these transmissions can begin.

The AIM list is used to keep track of overlapping active interactions in the DSS so that the one with the minimum speed requirement takes effect until interactions are completed. This list is updated when an interaction is completed or when a new interaction starts.

Broadcast of s-messages and use of the PIM lists enable distributed synchronous control of the simulation speeds for all logical processes. Figure 4-6 depicts a simplified version of the structure of an LSA and its position in the DSS architecture. A more detailed explanation of the LSA architecture is provided in the next chapter.

Figure 4-6: LSA structure and DSS architecture
Local synchronization agents can be automatically customized by using an interpreter that analyzes simulation models and generates all of the necessary structural models along with the look-up tables. Figure 4-7 depicts the process of creation of the structural models from simulation models. Obviously, the interpreter should be developed for the specific simulation package/language used to create simulation models. However, once an interpreter is developed it will reduce the development time of DSS significantly with this framework. In this research, a limited interpreter and a post-processor for the Arena™ simulation package were developed.

![Diagram](image)

**Figure 4-7: Synthesis of structural models**

Use of synchronization agents is a modular approach that eliminates the need to create custom simulation models for synchronization purposes. Simulation processes still need the functionality to communicate and interact in a distributed setting (such as compliance with HLA or some sort of client/server type communication capability). However, synchronization is isolated from the simulation process and it is the job of the synchronization agent, which runs as a separate process on the same computer with the simulation process.
In Chapter 5, the architecture of an LSA as well as its functioning and the overall functioning of a DSS are explained in full detail.
Chapter 5

Synchronization Implementation Framework

An adaptive partial pacing (APP) synchronization framework is developed with two principle objectives:

1. Rapid development and deployment of distributed simulations
2. Efficient synchronization of distributed simulations

Rapid development is addressed by automatically generating structural models from simulation processes as was discussed in Chapters 3 and 4. This chapter focuses on the synchronization framework and underlying algorithms used for the efficient synchronization of distributed simulations.

The chapter is organized as follows: the synchronization system architecture is explained in general in Section 1. Section 2 introduces and discusses the concept and components of the local synchronization agent (LSA) architecture. Section 3 discusses prediction of potential output messages (POMs). Section 4 discusses management of the message lists in the LSA. Section 5 presents an analysis for determination of an efficient update frequency for potential interactions during runtime. Section 6 presents the algorithms that form the Adaptive Partial Pacing (APP) framework used in this architecture. Finally, in Section 7, some concluding remarks regarding the general synchronization framework is presented.

5.1 Synchronization System Architecture

The main elements of the synchronization framework are the simulation processes and the synchronization agents that are linked to them. The framework is composed of a
distributed arrangement of these elements along with services necessary for paced synchronization in the DSS.

By definition, this framework does not provide any services (or suggestions) for partitioning a sequential simulation process into multiple distributed processes. In this research, it is assumed that the partitioning is already made and provided for this system. Therefore, any DSS created using this framework by default consists of $N$ sequential simulation processes (LPs), which have the inherent capability of communicating with each other.

Each simulation process is paired with a local synchronization agent (LSA). Local synchronization agents are the core of the framework, and they contain all of the necessary services for synchronization. LSAs communicate with each other by passing certain messages to initiate and maintain synchronization in the DSS.

In order to distinguish the messages that the simulation processes use to interact among each other from the set of messages that LSAs use for synchronization, a message classification is made. This classification is summarized in Table 5-1.
<table>
<thead>
<tr>
<th>Message Class</th>
<th>Member Message Objects</th>
<th>Object Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>i-message</td>
<td>MESSAGE</td>
<td>Messages that LPs send between each other to interact.</td>
</tr>
<tr>
<td>s-message</td>
<td>POM</td>
<td>Predicted output messages. These are replicas of the messages in the “i-message” class with extra information added (such as predicted virtual timestamps) and used for tracking potential messaging between LPs.</td>
</tr>
<tr>
<td></td>
<td>CANCELPOM</td>
<td>This message notifies LSAs to remove a POM from the PIM list.</td>
</tr>
<tr>
<td>notice</td>
<td>READY</td>
<td>This message signals readiness of an LSA before execution begins. It is used to create the handshaking among the LSAs for the coordinated start of execution.</td>
</tr>
<tr>
<td></td>
<td>DONE</td>
<td>This message signals the end of a broadcast of series of POM messages. It also includes a checksum variable which is used to validate the receipt of all broadcasted POM messages by the receivers.</td>
</tr>
<tr>
<td></td>
<td>ACK</td>
<td>This message is used to acknowledge readiness of a potential output message becoming an active output message.</td>
</tr>
<tr>
<td></td>
<td>ENDSYNC</td>
<td>This message is used to coordinate the de-activation of an active message after the respective interaction is completed.</td>
</tr>
<tr>
<td></td>
<td>STARTSYNC</td>
<td>This message is used to synchronously activate a potential output message.</td>
</tr>
</tbody>
</table>
The “i-message” class is the parent class of all MESSAGE objects. MESSAGE objects represent the type of messages passed between simulation processes in order to interact properly. These are the main players of the distributed simulation and issued/sent/received by simulation processes. LSAs do not manipulate these messages directly. The “i-message” class is defined in Table 5-2.

Table 5-2: Definition of the “i-message” class

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Attribute Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>Unique identifier of the message object.</td>
</tr>
<tr>
<td>Entity</td>
<td>Entity issuing the message object.</td>
</tr>
<tr>
<td>OriginLP</td>
<td>LP issuing the message object.</td>
</tr>
<tr>
<td>TargetLP</td>
<td>LP receiving the message object.</td>
</tr>
<tr>
<td>Timestamp</td>
<td>Scheduled virtual time of execution of the remote event carried by the message object.</td>
</tr>
<tr>
<td>VIL</td>
<td>Virtual interaction lag of the message object.</td>
</tr>
</tbody>
</table>

The “s-message” class is the parent class of all predicted/potential output messages (POMs). POMs are used to keep track of potential “i-messages” before they are fired and during their transmission. POMs are directly issued by LSAs and passed among LSAs to coordinate synchronization of LPs. The “s-message” class definition is provided in Table 5-3.
The “notice” class is the parent class of three message types: READY, DONE and ACK. The READY message object is broadcasted during initialization to notify other LSAs that the issuing LSA is ready for execution. The DONE message object is broadcasted after a series of POMs are broadcasted to notify other LSAs that the issuing LSA has completed sending POMs and also used for error-checking at the receiving LSAs. The ACK message object is broadcasted to acknowledge registration of a POM as an active message. These messages and their use are discussed in more detail in Section 5.6, where the APP algorithms are presented. The “notice” class definition is provided in Table 5-4.
The synchronization system functions in a distributed fashion. There is no centralized control in this framework, which provides more efficient communication by eliminating an additional central synchronization agent that receives and broadcasts all information.

Although a central controller would benefit this system by eliminating the need to store necessary synchronization data at each local synchronization agent, it creates an additional load on the communication network since an extra node is involved in all communications for synchronization.

The general architecture of the synchronization framework is depicted in Figure 5-1. Local synchronization agents act as a wrapper of the coupled simulation processes. Although a simulation process runs as an independent process on the same computer as the coupled local synchronization agent, the synchronization agent controls the advance of the simulation process. Therefore, at all times during an execution, simulation processes are under the control of local synchronization agents.
In this architecture, a dedicated communication network is utilized for synchronization communication and another dedicated communication network is used for simulation interactions. However, this is not a necessity and both synchronization agents and simulation processes can use the same network infrastructure if the network bandwidth is large enough to carry both message traffic. Nevertheless, considering the low cost of installing a local area network, it is advantageous to separate the message traffic for synchronization from the message traffic for simulation interactions.

In this divided communication architecture, messages from class “i-message” transmitted only in Dedicated Communication Bus B, while messages from classes “s-message” and “notice” are transmitted only in Dedicated Communication Bus A.

Briefly, a local synchronization agent’s job is to: 1) read the state of the entities in the coupled simulation process, 2) store this information in the entity-specific structural models to keep track of the progress of the simulation process, 3) predict the timing of the approaching potential outgoing messages, 4) broadcast this information to other synchronization agents, and 5) pace the coupled simulation process based on the collective information formed in the distributed simulation system.
The functions/services embedded in a local synchronization agent are as follows:

1. Reading states of active entities in the simulation process.
2. Creation and maintenance of entity-specific structural model objects for active entities of the simulation process.
3. Tracking the state of entities in the simulation process by using entity-specific structural model objects and entity-state information obtained from the simulation process.
4. Keeping statistics related to the firing of output messages for adjusting certain parameters of the synchronization system to improve its performance as the simulation process reaches steady-state.
5. Determining potential output messages (POMs) and periodically calculating minimum virtual time until their firing.
6. Maintaining a list of all locally originating potential output messages (POM) along with their timestamps and broadcasting this information to other synchronization agents.
7. Receiving POMs from other synchronization agents and maintaining a list of imminent interactions in the DSS.
8. Calculating the time-scaling factor for interactions and adjusting the speed of the coupled simulation process during interactions.
9. Synchronizing the system clock of the host computer with other host computers in the system (or with a higher tier clock provider such as a GPS receiver or a network time server).

In the next section, the components of the LSA and how above listed functions and services are embedded in the components are discussed in detail.
5.2 Components of a Local Synchronization Agent

A local synchronization agent has four interacting components that provide the functions and services introduced in the previous section. Figure 5-2 depicts the anatomy of a local synchronization agent. A synchronization agent is composed of two main components and two helper components:

1. Interaction Management Engine (IME)
2. Simulation Control Engine (SCE)
3. Timing and Clock Synchronization Module (TCSM)
4. Communication Module (CM)

---

Figure 5-2: Anatomy of a local synchronization agent
5.2.1 The Interaction Management Engine (IME)

The interaction management engine (IME) is the core of the synchronization agent and performs the most critical functions regarding the synchronization of the simulation processes. The IME is responsible for the following functions:

a. Creation, maintenance and disposal of entity-specific structural model objects.
b. Prediction of potential outbound messages and their timings.
c. Maintenance of the Predicted Output Messages (POM) list.
d. Maintenance of the Predicted Interaction Messages (PIM) list.
e. Maintenance of the Active Interaction Messages (AIM) list.

POM, PIM and AIM lists are the major data structures used to keep track of potential and active interactions in the DSS. The interaction management engine receives entity-state update information from the simulation control engine (SCE) and updates states of all active entities in their respective entity-specific structural model objects. If a new entity is created in the simulation, the IME creates a new entity-specific structural model object of the entity type.

One of the important attributes of an entity-specific structural model object is its state-tables. Every state in an entity-specific structural model object is attached a state-table. This table holds the amount of minimum virtual time to reach every accessible output event that fires an output message.

When a state update is performed on an entity-specific structural model object, the IME examines the timing table of the new state and creates a list of potential output messages. This operation is continuously performed for every active entity as the local simulation process executes.

The IME gathers all potential output messages from all active entity-specific structural model objects and adds it to the predicted output message (POM) list. The
POM list contains all locally originating potential output messages. The information in the POM list must be shared with all other LSAs in the DSS, because the synchronization framework depends on coordinated pacing of simulation processes based on these predicted messages. Therefore, every LSA must broadcast its POMs and a collective awareness of all potential interactions in the DSS must be formed and maintained throughout execution.

In this respect, a key function of the IME is to predict output messages before they fire, register them in the POM list (or erase them from the POM list when they become inaccessible or obsolete) and periodically send this information to the communication module to be broadcasted to all other LSAs.

In a complex simulation with many active entities, the POM list would contain several potential messages. It may not be efficient to broadcast all of this information periodically. Also, the frequency of these broadcasts will affect the performance of the overall synchronization system. Ideally, the more frequent the updates, the more efficient the system will work since inaccessible messages will be detected earlier and timing predictions will be corrected faster as the simulations advance.

However, there is a trade-off between frequent updates and the communication load on the communication network. If the bandwidth of the communication network is not infinite (which is a realistic assumption), as the number of active entities increase in the system (and also as the complexity of simulation processes increase – creating many alternative paths to many possible output events), updating the information related to potential output messages as frequently as they change will create an overload in the communication network.

Considering that communication is usually the main bottleneck in most distributed systems, a balance between the number of POMs to be broadcast, frequency of these broadcasts and the network load must be reached. In order to determine this
balance, a priority allocation mechanism is developed to dynamically determine the broadcast patterns. Details of this mechanism are discussed in Section 5.5.

The IME is also responsible for retrieving POMs broadcasted by other synchronization agents. POMs originated at remote LSAs are received by the communication module (CM) and forwarded to the IME. Upon receipt, IME stores remote POMs in the potential interaction messages (PIM) list. PIM list also includes locally originating POMs which are broadcasted in the DSS. Consequently, the PIM list contains all approaching (local and global) interactions in the DSS.

The IME continuously monitors the PIM list and when the local virtual time reaches the predicted timestamp of the first POM in the PIM list (PIM list is sorted in increasing timestamp order), it immediately forwards the corresponding interaction information to the simulation control engine (SCE) and moves the interaction message to the active interaction messages (AIM) list. The activated POM stays in the AIM list until the respective interaction in the DSS is completed. Completion of an interaction is signaled by the ENDSYNC message broadcasted upon receipt of the message.

The purpose of the AIM list is to keep track of active interactions in the DSS in case a new interaction with a stricter speed requirement takes effect before a previously started interaction is completed. In this case, the IME updates the SCE and also the AIM list so that the new speed requirement is implemented by the SCE.

If there are many active interactions in the DSS and one of them gets completed, the IME updates the AIM (by erasing the respective active interaction) and finds the active interaction in the AIM with the next strictest speed requirement and immediately passes the speed information to the SCE. This operation is performed synchronously at all LSAs at a wallclock time decided by the receiving LP’s synchronization agent. Consequently, after an interaction is completed all simulation processes change their
execution speeds within a reasonably short time period without creating a large gap between their local virtual clocks.

5.2.2 The Simulation Control Engine (SCE)

The simulation control engine (SCE) is responsible for monitoring the active entities of the coupled simulation processes and controlling their execution. The SCE is the only component that directly engages with the simulation process that is a totally separate program running on the same host computer as the LSA. Therefore, the SCE implementation must utilize inter-process communication and control tools that are specific to the operating system of the computer running the simulation process and the LSA.

The SCE receives pacing information from the IME during execution and makes sure that the local virtual clock of the coupled simulation process does not advance beyond the value of the pacing timer. The actual implementation of the speed control would be specific to the simulation software platform running the simulation process. For efficient control and inter-process communication, some functions of the SCE can be embedded in the simulation software (if the simulation software allows customization).

Monitoring of the active entities in the simulation process is also done by the SCE. This can be either done by active intervention of the SCE in the simulation process (by reading a state table or other recorded information in the memory) or passively by passing information from the simulation process to the SCE. SCE uses this information to update the structural model objects in the IME.
5.2.3 The Timing and Clock Synchronization Module (TCSM)

The timing and clock synchronization module (TCSM) is an auxiliary component in the LSA which provides timing service to the SCE for accurate and coordinated pacing of the local virtual clock. The timing service plays a central role in the accuracy of the overall synchronization service since independent LSAs depend on this service to coordinate the distributed pacing activity.

Proper coordination by timing requires the system clocks of host computers be in agreement in terms of both value and frequency. Therefore, the TCSM is also responsible for synchronizing the hardware clock of the host computer with an external reference clock. The reference clock can be a server computer providing time service in the local network (i.e., a time server), or it can be a truly external timing source such as a global positioning satellite (GPS).

The de facto industry standard for clock synchronization in computer networks is the Network Time Protocol (NTP) and a detailed discussion of NTP can be found in [Mills, 1994]. NTP is currently supported and embedded in Microsoft Windows XP and several Linux distributions including Red Hat v9.0. Therefore, using NTP for clock synchronization is fairly easy to implement in this framework.

5.2.4 The Communication Module (CM)

Last but not least, the communication module (CM) of the LSA is the auxiliary component that is responsible for all inter-LSA communication in the DSS. The CM has an input and an output buffer for temporarily holding incoming and outgoing messages. It has several stand-alone processes that run continuously in the background during execution to accept incoming messages and act according to the received message type.
Section 5.6 discusses the algorithms developed to implement the above mentioned services and functions of the components of the LSA. These services are called auxiliary services since they do not directly accomplish the synchronization, but help the core synchronization service create and maintain synchronization in the DSS.

5.3 Prediction of Potential Output Messages

In Chapters 3 and 4, structural models and their automated generation from simulation models were discussed. Structural models are finite state machine representations of simulation models and the fundamental reason that they are used in this synchronization framework is to keep track of simulation entities and predict the timing of potential output messages.

Prediction of the POM timings lies in the heart of the developed synchronization framework. The synchronization algorithm depends on these predictions to be ready for pacing individual simulation processes before an actual message/interaction becomes active in the DSS.

If the pacing of execution starts after the respective message is transmitted, there is no guarantee that the synchronization will be protected. Therefore, the synchronization algorithm must proactively pace the execution of the simulation processes for every interaction in the DSS.

In the case that the simulation process that is sending output messages is completely deterministic (i.e., all delays in the simulation process are known constants), then it is fairly easy to predict the timing of output messages by adding up state delays. However, this is rarely the case. Most analysis-intended simulation processes are stochastic and precise timing of output messages cannot be known in advance.
In order to resolve this issue, simulation processes to be synchronized using the developed synchronization framework must adhere to the following rule:

**Rule 5.1:** Every random-valued delay in a message-sending simulation process must have a positive constant lower bound.

This rule guarantees that, there exists a positive lower bound on the amount of virtual time elapsed from the creation of an entity to the firing of an output message for all possible output messages in the DSS.

If Rule 5.1 is satisfied by all message-sending simulation processes in a DSS, the algorithm can predict the earliest possible virtual time of firing of an output message and proactively start pacing when LVC reaches that virtual time point.

Obviously, not all potential messages will become actual messages and as entities progress through a simulation process, the timing predictions will change based on the actual amount of virtual time spent in previous states. Therefore, predictions must be updated periodically to account for these changes occurring during runtime.

Updating timing predictions corresponds to making new predictions at the current state of an entity. Thus, the synchronization algorithm must know the least amount of virtual time it takes an entity to reach an output event from every state in the simulation process. While Rule 5.1 is in effect, the following method can be used for predictions:

**Method 5.1:** To predict the minimum amount of virtual time required to reach an output event $e_i$ from state $s_k$: find the total cost of the shortest path from state $s_k$ to a state $s_j$, where each state in the entity-specific structural model object has a cost that is equal to the minimum virtual time delay in that state, and $e_i = (s_i, s_j)$ where $s_j$ is a final state.
This method calculates the least amount of time it will take for an entity to reach the final state right after firing an output event/message (note that firing of an event is instantaneous and all delays occur at states). Since all of the required information to execute Method 5.1 is known in advance before the runtime, these calculations can be performed before the runtime for every state in every entity-specific structural model class and resulting timing predictions can be stored in state-tables to be used during runtime. This scheme provides a fast way of updating the predictions during runtime where processing power is more valuable by only performing a quick linear search in state-tables.

“Dijkstra’s Shortest Path Algorithm” is used to calculate the cost of the shortest path. This algorithm is suitable since all costs (minimum state delays) in a structural model are positive. A detailed explanation and a mathematical formulation of the shortest path problem as well as the “Dijkstra’s Shortest Path Algorithm” can be found in [Bazaraa et al., 1990]. Here is the algorithmic form the “Dijkstra’s Shortest Path Algorithm” to be used in this framework.

Let $S = (X, \Sigma, \Gamma, \Psi, \bar{X})$ be a state machine with $m$ states (i.e. $|X| = m$) and $n$ events (i.e. $|\Sigma| = n$). Each state $\chi_i$ has a cost (minimum virtual time delay) $c_i \geq 0$. Let $\chi_0$ be the starting state and $\chi_m$ be the final state for the shortest path calculation.
**Initialization:**
Define and set:

- \( \omega_i = c_i, \quad \forall \chi_i \in \overline{X} \) (i.e. all initial states) and \( \omega_i = 0, \quad \forall \chi_i \notin \overline{X} \)
- \( Z = \{ \chi_0 \} \)
- \( A = \emptyset \)
- \( m = \|X\| \)
- \( k = 0 \)

**Main Procedure:**

Step 1. Set \( \overline{Z} = X - Z \)
Step 2. Set \( A = \{ \sigma_{ij} : \chi_i \in Z, \chi_j \in \overline{Z} \} \)
Step 3. Find the states \( \chi_p \in Z \) and \( \chi_q \in \overline{Z} \) such that \( \omega_p + c_q = \min \{ \omega_i + c_j \} \)
Step 4. Set \( \omega_q = \omega_p + c_q \) and \( Z = Z \cup \{ \chi_q \} \)
Step 5. Set \( k = k + 1 \)
Step 6. If \( k = m - 1 \) then GOTO Step 7, else GOTO Step 1.
Step 7. Result = \( \omega_m \)
Step 8. END

This is a simple procedure and actually once it completes, the \( \omega_i \) value for any state \( \chi_i \) gives the total cost of the shortest path from the starting state to the state \( \chi_i \). Therefore, running this procedure once for a particular state is sufficient to find the minimum amount of virtual time required to reach all possible output events from that state.

The shortest path procedure is executed once for each state of an entity-specific structural model class and the resulting minimum virtual time values are stored in **state-tables**. These state-tables are used during run-time to determine the minimum virtual
time to reach a certain potential message in the POM list. Since the table look-up operation is very fast, the timing predictions for potential output messages in the POM list can be updated as frequent as after every state transition of active entities.

5.4 Management of the Message Lists

The POM list is the primary data structure where local POMs are stored during runtime. There are three cases that require an update of the information contained in a POM list:

**Case 1:** A new entity is created and the respective entity-specific structural model class has a non-empty output event set.

**POM Update Action:**
1. Identify the initial state of the new entity.
2. Retrieve the minimum time values for all accessible messages at the initial state from the state-tables.
3. Form the POMs according to the retrieved information in (2) and append them to the POM list.

**Case 2:** An existing potential output message in the POM list is no longer reachable by the associated entity.

**POM Update Action:** Remove the canceled message from the POM list.

**Case 3:** Timestamp of an existing potential output message in the POM list is increased due to an entity state transition.

**POM Update Action:** Update the timestamp for the message in the POM list.

Auxiliary services continuously check for these three cases and execute necessary POM update actions as listed above to manage the POM list.
The active input message (AIM) list is only used when a message becomes active and requires very little management. When a potential output message in the PIM list becomes active (i.e., the predicted virtual time of firing is reached) it is removed from the PIM list and inserted into the AIM list. After an active message existing in the AIM list is received by the target LP, the target LP sends an ENDSYNC message signaling the completion of the interaction and when this message is received by remaining LSAs, that particular active message is removed from the AIM list.

The predicted input messages (PIM) list contains potential output messages not only targeted for the local LP but also for all other LPs. Therefore, this list serves as a global watch-list for all interactions in the DSS. Proper management of the PIM list is very important for proper functioning of the synchronization algorithm.

Ideally, every time a local POM list is updated in the DSS, the PIM lists should be updated. This would yield the maximum synchronization algorithm performance. However, unlike the management of the POM list, it may not be possible to update the information present in the PIM list as frequent as the ideal case. Every update requires dissemination of the new information to all LSAs by broadcasting POMs via the communication network.

Therefore, an efficient updating scheme is developed and details of this scheme are discussed in the next section.

5.5 Determination of an Efficient Update Frequency for the PIM List

During an execution, every simulation process that is capable of firing output messages (i.e., passing entities to other simulation processes) drives its local synchronization agent to populate its POM list. Before a potential message becomes an active message (i.e., an interaction), the synchronization algorithm starts paced execution
for that interaction so that the synchronization property is conserved during the interaction. Figure 5-3 illustrates the lifecycle of an interaction.

Each period in the lifecycle is depicted with a color:

- **Period A (red)** corresponds to interaction-free (not paced) execution.
- **Period B (blue)** corresponds to the error made in predicting the timing of the firing of the output message.
- **Period C (orange)** corresponds to the actual interaction – transmission of the output message from the source LP to the receiver LP.
- **Period D (green)** corresponds to the time it takes the receiver LP’s LSA to acknowledge other LSAs that the output message is received.
- **Period E (yellow)** corresponds to the time consumed for coordinated switching of the speed. The leader (receiver LP) decides a time instant at which the current interaction speed becomes obsolete and includes this information in the acknowledgement message mentioned above. This is necessary since the speed change must be done synchronously.

The performance of the synchronization algorithm depends on the length of the periods B, D and E. The DSS need not be in the low interaction speed during these periods, because the real interaction takes place only during period C.
Lengths of periods D and E are fairly uncontrollable because they depend on the speed of message communication. Therefore, the length of period B can be the single most important controllable factor affecting the performance of the APP algorithm. For this reason, accurate prediction of the message firing time is important, which depends on the requirement of frequently updating POM timestamps.

Naturally, this requirement for frequent broadcasts of information by many LSAs during execution creates the need for a high bandwidth communication network for the LSAs. Failing to have one and allowing unlimited and uncontrolled broadcasting according to arbitrary local policies would eventually create overloading of the communication network and cause degradation of the synchronization algorithm’s performance.

One strategy for allocating the available bandwidth among LSAs is to simply allocate them equally. This strategy is easy to implement, however it will perform well only if output message firing frequencies of all LPs are reasonably uniform. In a stochastic simulation system this will not be always true.

Therefore, a better strategy is to dynamically allocate the bandwidth among the LSAs with priority given to those whose LPs are closest to sending actual messages. This basically means that the bandwidth should be allocated based on the information contained in the PIM lists.

Let $B$ be the theoretical bandwidth of the dedicated communication network used for synchronization. Due to inefficiencies in the network such as processing of the packets in the hardware, the actual bandwidth is usually less than the theoretical bandwidth. This is called the efficient bandwidth of the network and it is denoted by $B_{\text{eff}}$, which is a fraction ($0 < \alpha < 1$) of the theoretical bandwidth:
Although the efficient bandwidth depends on many parameters and can change dynamically for a general wide area network, when a dedicated local area network is considered (such as the one used in this framework) the efficient bandwidth can be treated as a known constant.

First and foremost, a minimum amount of bandwidth must be allocated to each LSA regardless of its priority in terms of message updates to allow for transmission of mission-critical messages:
1. Host computer system clock synchronization messages.
2. Messages of the “notice” class.
3. New potential output messages due to new entity creation.

The above listed basic communication requirements create a base level of bandwidth, $q_{\text{base}}$, to be allocated to every LSA.

The amount of the base allocation increases as the number of simulation processes increases. Therefore, the amount of the efficient bandwidth must be greater than the base requirement ($N$ is the number of LPs):

$$B_{\text{eff}} > N \cdot q_{\text{base}}$$  \hspace{1cm} (5.2)

In fact, the efficient bandwidth must be substantially larger than the base requirement so that, a large quantity of bandwidth is available for frequent updates. Assuming that Eq. 5.2 holds, the remaining bandwidth after the base allocation is:

$$B_{\text{rem}} = B_{\text{eff}} - N \cdot q_{\text{base}}$$  \hspace{1cm} (5.3)

Fair allocation of the remaining bandwidth $B_{\text{rem}}$ must be decided based on the factors that affect the performance of the synchronization algorithm. Since, the performance of synchronization depends on the error made in predicting the virtual time
required for the firing of the output messages, at least part of the objective of an efficient/fair allocation policy must be to minimize the prediction error.

Prediction error is a complex function of the simulation process itself, since it depends on the actual virtual time spent by entities in simulated states. The synchronization algorithm uses lower limits on the delay at each state to predict the virtual time of firing of an output event. For any complex stochastic simulation, it is not possible to create an accurate mathematical model for the prediction error. Consequently, it is also not possible to optimize the allocation of the remaining bandwidth based on a mathematical analysis. However, it is possible to create an allocation policy that dynamically changes the allocation of the bandwidth towards the advantage of the LSAs that are creating large prediction errors and lagging in terms of synchronization efficiency.

To this extent, an allocation policy, which can be implemented in a distributed manner based on an indexing scheme, is devised. An “Allocation Priority Index” (API) is formed to measure the relative allocation priority of each “potential message-LSA” pair during runtime. API is calculated for each potential output message in the PIM list and a higher API value means more bandwidth to be allocated to that LSA for updating the timing of that particular potential output message present in the PIM list.

Three parameters are used to calculate the API:

- \( R_i(T) \): Virtual time residual for message \( i \) at virtual time \( T \).
- \( \bar{e}_i(T) \): Average prediction error for message \( i \) up to virtual time \( T \).
- \( p_i(T) \): Likelihood of firing of message \( i \) predicted based on the previous data up to virtual time \( T \).
Among these three parameters, the virtual time residual, $R_i(T)$, is the most important one, since it measures the remaining amount of virtual time until the predicted firing of a message. It is calculated as:

$$R_i(T = T_{NOW}) = \tilde{T}_i - T_{NOW}$$ (5.4)

Here, $T_{NOW}$ is the current virtual time (during runtime) and $\tilde{T}_i$ is the predicted virtual time of firing for POM $i$. $R_i(T)$ is directly proportional to the index of the potential output message $i$ in the PIM list. The closer the predicted virtual time of firing of a message, the higher it is positioned in the PIM list. Naturally, more frequent updates should be allowed for more imminent messages since other messages positioned lower in the PIM list have more time for updates until they fire.

The average prediction error parameter, $\overline{e}_i(T)$, is calculated based on past observations for each output message at the hosting LSA and is embedded in a POM as an attribute. Every time a certain output message fires, its actual virtual time of firing and its predicted virtual time of firing from the PIM list is temporarily recorded by the Interaction Management Engine (IME) to calculate the $\overline{e}_i(T)$ for that output event.

The IME monitors firing of each output event, which actually corresponds to transportation of an entity between simulation processes. Thus, $\overline{e}_i(T)$ actually depends directly on the departure process of that particular entity in the coupled simulation process. Considering a sufficiently long simulation execution for well behaved complex systems, the firing times for a simulation converge to a steady-state process. In fact, the purpose of a simulation analysis is to predict particulars of these complex stochastic processes at steady-state.

As the coupled simulation process reaches steady-state (if one exists), it is expected that $\overline{e}_i(T)$ will reach a steady-state value, and it is quite reasonable to use $\overline{e}_i(T)$ as an indicator of the error margin in predicting the virtual time of firing of a particular
output event. The aim of the developed allocation scheme is to minimize the collective error in making predictions, so that as little as required time is spent in a low-speed state during an execution.

Similar to the average prediction error, the likelihood of firing of an output event, \( p_i(T) \), is predicted by recording how many times a particular message actually fired after being placed in the PIM list and scaling this quantity by the total number of times that particular output message is placed on the PIM list. This is a very simple indication of the likelihood of firing of a POM present in the PIM list. Like \( \overline{e}_i(T) \), \( p_i(T) \) is also embedded in a POM as an attribute.

Both \( \overline{e}_i(T) \) and \( p_i(T) \) are not as reliable as \( R_i(T) \) in calculating the API since they are only predictions based on past behavior. On the other hand, a high \( R_i(T) \) indicates a need for frequent updating of a POM.

The API is calculated as follows:

\[
API_i(T) = \beta_1 f(R_i(T)) + \beta_2 g(\overline{e}_i(T)) + \beta_3 p_i(T)
\]

(5.5)

Here, the constants \( \beta_1 \), \( \beta_2 \) and \( \beta_3 \) are weights in the range \([0, 1]\) and they add up to 1. Values of these constants are to be adjusted according to the performance characteristics of the synchronization algorithm for a specific DSS. However, since \( R_i(T) \) dominates \( \overline{e}_i(T) \) and \( p_i(T) \), \( \beta_i \) should be greater than \( \beta_2 \) and \( \beta_3 \).

Functions \( f \) and \( g \) are transformations used to scale down values of \( R_i(T) \) and \( \overline{e}_i(T) \). They are defined as follows:

\[
f(R_i(T)) = e^{ \frac{(R_i(T) - R_0)^2}{2\sigma^2} }: R_M, \sigma \in (0, \infty)
\]

(5.6)

\[
g(\overline{e}_i(T)) = \frac{(\overline{e}_i(T) - c)^2}{c^2 + (\overline{e}_i(T) - c)^2} : c \in (0, \infty)
\]

(5.7)
The shape of the transformation $f$ (Eq. 5.6) is depicted in Figure 5-4. This transformation not only scales down $R_i(T)$ values into the range $[0, 1]$, but also amplifies them in the range $[R_M - 3\sigma, R_M + 3\sigma]$ while reducing the effect of the values outside this range. The sensitive range of the $f$ transformation is determined by the value of the spread parameter $\sigma$. This amplification is introduced into the calculation for the API due to the following reasons:

1. $R_i(T) < R_M - 3\sigma$: In this region the residual value is small due to the associated entity moving into a state that is very close to the output state. Therefore, at this point there is very limited information available to update the timing of a potential message and the weight returned by the $f$ transformation decreases rapidly.

2. $R_i(T) > R_M + 3\sigma$: In this region, the weight returned by the $f$ transformation is low since priority is given to those messages that are close to firing. Therefore, messages for which the residual value is in between the thresholds defined by this range have higher weights as determined by this transformation.

Figure 5-4: Shape of the $f$ transformation

The first reason stated above also affects $\overline{\xi}(T)$, such that the minimum prediction error has to be greater than the prediction error due to the delay at the last state before the firing of an output event. Therefore, the error at the last state must be discounted from the
calculation of $\bar{c}(T)$. This error value is denoted by $c$, and since it cannot be known ahead of time, it is estimated by taking the average of the observations for previously fired messages (i.e., actual virtual time spent in the last state less the minimum delay for that state).

The shape of the transformation $g$ (Eq. 5.7) is shown in Figure 5-5. As seen in this graph, the transformed function is sensitive to the increase in the value of $\bar{c}(T)$ at small values and the slope of the curve decreases as $\bar{c}(T)$ increases.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{g_transform.png}
\caption{Shape of the $g$ transformation}
\end{figure}

The $g$ transformation returns higher weight values as the error value increases, but the rate of increase of the weight decreases after a certain point since small error values are desired to be detected and penalized early to avoid occurrence of larger error values.

The API is calculated for every potential output message in the PIM list at every LSA. Since all PIM lists carry the same information, the same values will be calculated for every message at all LSAs. Once the API values of all messages in the PIM list are known, they are used as weights to calculate the fraction of the network bandwidth to be
allocated to each potential message. Thus, the amount of bandwidth allocated to potential message \( j \) is calculated as follows:

\[
q_j(T) = B_{\text{rem}} \cdot \frac{API_j(T)}{\sum_{\forall i} API_i(T)}
\]  \hspace{1cm} (5.8)

Every time a new potential output message enters the PIM list or a message is cancelled or the timing of a message is changed in the PIM list, the bandwidth allocation is redone. Since simple mathematical functions of real numbers are used, this is a very fast operation, which does not consume much processing power.

Once the allocated bandwidth to a POM is known, the update frequency \( \rho_i \) for that message is calculated by dividing the allocated bandwidth by the size of the update message \( s_i \):

\[
\rho_i(T) = \frac{q_i(T)}{s_i}
\]  \hspace{1cm} (5.9)

This dynamic allocation scheme avoids congestion in the network and shifts the allocation priority to the LSAs with imminent and more error-prone potential output messages.

5.6 The Adaptive Partial Pacing (APP) Algorithm

The adaptive partial pacing (APP) algorithm is the core algorithm running in all local synchronization agents. In the heart of the APP algorithm is the idea of creating an agreement and coordination among the distributed local synchronization agents in the DSS. The success of the algorithm strictly depends on this coordination, which is created by using synchronized real-time clocks (i.e., the system clock of the host computer). LSAs must start and end pacing of their coupled simulation processes at around the same wallclock time instants so that LVCs advance with almost the same pace during interactions.
This type of coordination requires one of the LSAs to provide the others with the timing information at every interaction. The LSA that provides the timing information is called the **leader**. The leader is determined for each interaction and it is the LSA that is coupled with the receiving simulation process for that interaction. This is due to the fact that beginning of an interaction is predicted by all LSAs, however completion of an interaction is only known by the receiver LP’s LSA. The leader LSA is responsible for informing other LSAs of the completion of the respective interaction. The completion message includes the timing information regarding when to switch to another pacing speed, so that the speed change is done synchronously.

One other issue that should be considered during interactions is the nesting of interactions. If two or more interactions overlap each other, this situation needs to be managed accordingly so that the speed requirements for all of these active interactions are taken into consideration.

For example, the DSS starts pacing for interaction A and before the completion messages for the interaction is broadcast by the receiving LP’s LSA, another interaction, interaction B becomes active. In this case, if interaction B’s speed requirement is stricter than that of interaction A, the system must respond to this and reduce the execution pace until interaction B is over.

The nesting of interactions requires that the system is paced according to the active interaction that has the strictest speed requirement. Once an active interaction is completed, if there are other active interactions, the system must select the speed of the interaction with the next strictest speed requirement. If there are no active interactions left, then the system must go back to regular execution with no pacing.

As explained in previous sections, the AIM list (active interacting messages) is used to keep track of all active interactions in the DSS. It is important to note that an
active interaction does not necessarily correspond to an in-transmission message. An interaction becomes active when LVCs reach the predicted time of firing (i.e. timestamp) of the corresponding potential output message.

The predicted time is the earliest time a particular output message can fire. Although, this prediction is updated periodically by the leader LSA, there is no guarantee that it will match the actual firing time of the respective output message. In fact, it is almost impossible to predict the exact time of firing for an output message when the originating LP is stochastic.

Although the APP algorithm is suitable for synchronizing different simulation models without being altered, it relies on the capability of LSAs to interact with the simulation process and control its execution. This functionality is implemented in the simulation control engine (SCE) of an LSA and depends upon the particular simulation platform used in a DSS.

The APP algorithm is a collection of separate procedures and functions which are contained in different components of an LSA. These procedures and functions are divided into two major categories:

- Auxiliary services and functions
- Core synchronization service

In the next two sub-sections, the algorithms that collectively form the APP algorithm are presented and described. Pseudo code and flowcharts are used to present the algorithms. Table 5-5 explains the pseudo-code notation and generic object/variable names used in the algorithm descriptions. Table 5-6 provides the legend for the flowcharts.
<table>
<thead>
<tr>
<th>Symbol/Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Keywords</strong></td>
<td></td>
</tr>
<tr>
<td>{</td>
<td>Begin – starts a procedure / statement.</td>
</tr>
<tr>
<td>}</td>
<td>End – terminates a procedure / statement.</td>
</tr>
<tr>
<td>function</td>
<td>Identifies a function.</td>
</tr>
<tr>
<td>service</td>
<td>Identifies a service.</td>
</tr>
<tr>
<td>if</td>
<td>If statement.</td>
</tr>
<tr>
<td>else</td>
<td>Else statement (always used after an if statement).</td>
</tr>
<tr>
<td>wait until</td>
<td>Wait until statement (waits until a condition is satisfied).</td>
</tr>
<tr>
<td>for</td>
<td>For statement (loops until a number of elements is processed).</td>
</tr>
<tr>
<td>while</td>
<td>While statement (Loops until a certain condition is satisfied).</td>
</tr>
<tr>
<td>obj.pro</td>
<td>Value of attribute PRO of object OBJ.</td>
</tr>
<tr>
<td><strong>Variables</strong></td>
<td></td>
</tr>
<tr>
<td>mode</td>
<td>Mode of execution = “Faced_Execution” or “Regular_Execution”</td>
</tr>
<tr>
<td>current_scaling_factor</td>
<td>The time scaling factor that is currently in effect for pacing.</td>
</tr>
<tr>
<td>lvc</td>
<td>Local virtual clock (of the coupled LP).</td>
</tr>
<tr>
<td>timer</td>
<td>A timer that is used to pace the LVC.</td>
</tr>
<tr>
<td>leader</td>
<td>Denotes the leader LP for a particular interaction.</td>
</tr>
<tr>
<td>lsa</td>
<td>The local synchronization agent.</td>
</tr>
<tr>
<td>systemclock</td>
<td>Current value of the system-clock or the system-clock itself.</td>
</tr>
<tr>
<td>delay</td>
<td>A value that is equal to a multiple of the max. transmission time.</td>
</tr>
<tr>
<td>syncperiod</td>
<td>The amount of time between system clock synchronizations.</td>
</tr>
<tr>
<td>timer_active</td>
<td>Activity status of the TIMER = “True” or “False”</td>
</tr>
<tr>
<td><strong>Objects</strong></td>
<td></td>
</tr>
<tr>
<td>nextevent</td>
<td>The next event in the event calendar of the coupled LP.</td>
</tr>
<tr>
<td>entity</td>
<td>A simulation entity.</td>
</tr>
<tr>
<td>current_active_message</td>
<td>The POM that represents the current interaction in the DSS.</td>
</tr>
</tbody>
</table>
### Table 5-6: Flowchart legend

<table>
<thead>
<tr>
<th>Class</th>
<th>Block</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Terminator</td>
<td><img src="image" alt="Begin" /></td>
<td>Starts a procedure</td>
</tr>
<tr>
<td></td>
<td><img src="image" alt="End" /></td>
<td>Terminates a procedure</td>
</tr>
<tr>
<td>Process</td>
<td><img src="image" alt="Predefined process" /></td>
<td>Predefined process</td>
</tr>
<tr>
<td></td>
<td><img src="image" alt="Process" /></td>
<td>Process</td>
</tr>
<tr>
<td>Decision/Loop</td>
<td><img src="image" alt="For(…)" /></td>
<td>For loop</td>
</tr>
<tr>
<td></td>
<td><img src="image" alt="Decision" /></td>
<td>Decision</td>
</tr>
<tr>
<td>Data</td>
<td><img src="image" alt="Data" /></td>
<td>Externally passed parameter/data</td>
</tr>
</tbody>
</table>
5.6.1 Algorithms of Auxiliary Services and Functions

Auxiliary services and functions were introduced and briefly discussed in Section 5.2 during discussion of the components of an LSA. In this section, the algorithms for these services and functions are defined and explained in detail.

A **service** is a procedure that runs continuously during a simulation execution. Auxiliary services perform tasks that are essential for proper functioning of the core synchronization service. On the other hand, a **function** is a procedure that is called from within a service or another function. Functions do not run continuously during an entire execution and parameters can be passed to functions.

Auxiliary services and functions are contained within different components of an LSA according to the tasks they perform. Figure 5-6 depicts the cross-component relational categorization of the auxiliary services and functions. Services are pictured using red blocks and functions are pictured using yellow blocks. Green links represent the dependency relationships between functions and services.
Figure 5-6: Cross-component relational categorization of auxiliary services and functions (red blocks – services; orange blocks – functions; green links - relations)
5.6.1.1 Communication Module (CM) Services and Functions

Communication module contains two services (EndSync and ReceivePIM) and two functions (Send and Broadcast). These services and functions accept incoming messages, act on incoming messages and send outgoing messages. They interact with the services and functions of the Interaction Management Engine (IME) and the Simulation Control Engine (SCE).

**EndSync service** (See Table 5-7 and Figure 5-7)

This service listens for incoming ENDSYNC messages. Once an ENDSYNC message is received, it waits until the system-clock reaches the timestamp of the received ENDSYNC message. Then, it erases the corresponding active message from the AIM list, meaning that the interaction represented by the erased message is over. After erasing the message, it calls the UpdateCAM function to update the current active message.

```plaintext
service EndSync()
{
    While (simulation is running)
    {
        If (an ENDSYNC message received)
        {
            NEWENDSYNC = ENDSYNC message in the CM input buffer;
            Remove NEWENDSYNC from the CM input buffer;
            Wait until (SYSTEMCLOCK >= NEWENDSYNC.Time);
            Remove NEWENDSYNC.Message from the AIM list;
            UpdateCAM();
        }
    }
}
```
Figure 5-7: The EndSync algorithm - flowchart
ReceivePIM service (See Table 5-8 and Figure 5-8)

This service listens for POM and CANCELPOM messages. When a POM is received it is placed in the PIM list. When a CANCELPOM message is received, the corresponding POM in the PIM list is deleted. In both cases, at the end, the UpdatePIM() service is activated if it is in the sleep mode.

```
service ReceivePIM()
{
    While (Simulation is running)
    {
        If (a POM message is received in the CM input buffer)
        {
            NEWPOM = received POM message;
            Remove NEWPOM from the CM input buffer;
            Append NEWPOM to the PIM list;
            If (UpdatePIM() is sleeping)
            {
                Wake up UpdatePIM();
            }
        }
        If (a CANCELPOM message is received)
        {
            NEWCANCELPOM = received CANCELPOM message;
            Remove NEWCANCELPOM from the CM input buffer;
            Remove the POM with POM.ID==NEWCANCELPOM.ID from the PIM list;
            If (UpdatePIM() is sleeping)
            {
                Wake up UpdatePIM();
            }
        }
    }
}
```
Figure 5-8: The ReceivePIM algorithm - flowchart
**Send and Broadcast functions** (See Table 5-9 and Figure 5-9)

These two functions simply transmit passed messages to the designated destination LSA(s). The *Broadcast* function uses the *Send* function in a loop to send the designated message to all LSAs in the DSS.

```
function Send(MSG, DEST)
{
    Transmit message MSG to destination LSA DEST;
}

function Broadcast(MSG)
{
    For (all LSAs)
    {
        CURRENTDEST = currently addressed LSA;
        Send(MSG, CURRENTDEST);
    }
}
```

---

*Figure 5-9:* (a) *Send* and (b) *Broadcast* algorithms – flowchart
5.6.1.2 The Timing and Clock Synchronization Module (TCSM) Services and Functions

The Timing and Clock Synchronization Module (TCSM) consists of two functions (StartTIMER and StopTIMER) and a service (ClockSync).

**ClockSync service** (See Table 5-10, Figure 5-10):

This service synchronizes the system clock of the host computer (i.e. SYSTEMCLOCK) with the external clock provider periodically. The details of the synchronization are not explained here since it depends on the method chosen to implement the synchronization (such as network time synchronization or global positioning system time synchronization). Time between two consecutive synchronizations are held in the SYNCPERIOD variable and changes according to the method of synchronization due to different synchronization accuracies.

```service ClockSync()
{
    While (Simulation is running)
    {
        Synchronize SYSTEMCLOCK;
        Last_Sync_Time = SYSTEMCLOCK;
        Wait until (SYSTEMCLOCK - Last_Sync_Time >= SYNCPERIOD);
    }
}
```
**StartTIMER and StopTIMER functions** (See Table 5-11 and Figure 5-11)

These two functions interact with each other by means of the shared variable `TIMER_Active`, which indicates the status of the `TIMER`.

The `StartTIMER` function first sets the `TIMER_Active` variable to `True`. While the `TIMER_Active` is `True`, it times one simulation time unit and increments the value of the `TIMER`. One simulation time unit is equivalent to one real-time unit multiplied by the scaling factor (see Chapter 3 for more detailed information).

The `StopTIMER` function sets the `TIMER_Active` variable to `True` and then sets the local virtual clock to the ending value of the `TIMER`. Calling the `StopTIMER` function while the `StartTIMER` function is looping causes the `StartTIMER` function to exit the loop.
```plaintext
function StartTIMER()
{
    TIMER_Active = True;
    While (TIMER_Active == True)
    {
        Start_Time = SYSTEMCLOCK;
        Elapsed = SYSTEMCLOCK - Start_Time;
        Wait until (Elapsed >= Current_Scaling_Factor*Real-time unit);
        Increment TIMER;
    }
}

function StopTIMER()
{
    TIMER_Active = False;
    LVC = TIMER;
}
```

Figure 5-11: (a) StartTimer and (b) StopTimer algorithms – flowchart
5.6.1.3 The Simulation Control Engine (SCE) Services and Functions

The simulation control engine (SCE) contains two services (\texttt{UpdateSMC} and \texttt{EndSyncL}) and two functions (\texttt{StartPacing} and \texttt{StopPacing}).

\textbf{UpdateSMC service} (See Table 5-12 and Figure 5-12)

This service continuously updates the structural model core of the interaction management engine (IME). There are too main cases: (1) a new entity is created in the simulation process and (2) an entity changed its state in the simulation process. The information related to entities is supplied to the simulation control engine by the simulation process. The method of supplying entity update information from the simulation process is simulation software dependent and therefore not discussed here.

In case (1), this service only acts if the simulation is not at the very beginning of the execution (i.e. LVC = 0), because the structural model core is initialized at the beginning by the \texttt{Initialize} sub-procedure.

When a new entity is created, this service first creates a new structural model object matching the type of the new entity. After that, it identifies the initial state of the newly created entity and inserts all POMs in the initial state-table of the structural model object to the POM list.

In case (2), there are two sub-cases: (a) the entity arrived at a final state and disposed by the simulation process, and (b) the entity arrived at a non-final state. In the first sub-case, the service cancels all existing POMs of the entity, updates the related statistics for the entity type and deletes its structural model object. In the second sub-case, the service first updates the state of the entity in its structural model object. Since the entity moved to a new state, the service iterates through all existing POMs of the entity and updates their timestamps in accordance with the information in the new state-table or if an existing POM is not present in the new state-table, the service cancels it.
```java
service UpdateSMC()
{
    While (Simulation is running)
    {
        If ((a new entity is created) AND (LVC != 0))
        {
            NEWENTITY = the new entity;
            Create new entity-specific structural model object of the type NEWENTITY.Type;
            For (all POMs in the initial state-table of NEWENTITY)
            {
                NEWPOM = currently addressed POM in the initial state-table of ENTITY;
                NEWPOM.Status = Current;
                Append NEWPOM to the POM list;
            }
        }
        If (an entity state change occurred)
        {
            ENTITY = the entity that changed state;
            If (ENTITY arrived in a final state and disposed)
            {
                Remove all POMs of the ENTITY from the POM list;
                Update the statistics for the entity-type ENTITY.Type;
                Delete ENTITY's entity-specific structural model object;
            }
            Else
            {
                Update entity-specific structural model object of ENTITY;
                For (all POMs of the ENTITY in the POM list)
                {
                    CURRENTPOM = currently addressed POM of the ENTITY in the POM list;
                    If (CURRENTPOM exists in the new state-table of ENTITY)
                    {
                        Update CURRENTPOM.Timestamp according to the new state-table;
                        CURRENTPOM.Status = Updated;
                    }
                    Else
                    {
                        Remove CURRENTPOM from the POM list;
                    }
                }
                For (all POMs in the new state-table of ENTITY)
                {
                    CURRENTPOM = currently addressed POM in the new state-table of ENTITY;
                    If (CURRENTPOM does not exist in the POM list)
                    {
                        Append CURRENTPOM to the POM list;
                    }
                }
            }
        }
    }
}
```
Figure 5-12: UpdateSMC algorithm – flowchart
**StartPacing and StopPacing functions** (See Table 5-13 and Figure 5-13)

These two functions are used to start and stop paced execution. The `StartPacing` function first initializes the `TIMER` by setting it to the current value of the local virtual clock (LVC). Then it calls the `StartTIMER` function that starts the `TIMER`. The `StopPacing` function simply calls the `StopTIMER` function to stop the `TIMER`.

```plaintext
function StartPacing()
{
    TIMER = LVC;
    StartTIMER();
}

function StopPacing()
{
    StopTIMER();
}
```

---

**Figure 5-13:** (a) `StartPacing` and (b) `StopPacing` algorithms – flowchart
EndSyncL service (See Table 5-14 and Figure 5-14)

This service monitors the coupled simulation process (LP) for receipt of an interaction message. Once a message is received (i.e., an active interaction is completed), it finds the corresponding POM in the AIM list and broadcasts an ENDSYNC message since the LSA acts as the LEADER for this particular interaction. The ENDSYNC message carries the identifier for the POM and a time value which signals the other LSAs at what instant this particular interaction is officially over. This is used to synchronously change the mode or speed of execution in the DSS.

The ending time of an interaction is calculated by adding a constant delay amount (i.e. value of the Delay variable) to the current value of the system-clock. The delay amount is a multiple of the maximum transmission delay in the network and it is used to guarantee the receipt of the ENDSYNC message before the ending time is reached.

After the ending time is reached, the service removes the obsolete POM from the AIM list and updates the current active message.

```
service EndSyncL()
{
    While (simulation is running)
    {
        If (a MESSAGE is received by the LP)
        {
            NEWMESSAGE = MESSAGE received by the LP;
            AIM_MESSAGE = POM in the AIM list for the NEWMESSAGE;
            End_Time = SYSTEMCLOCK + Delay;
            ENDSYNC.Message = AIM_MESSAGE;
            ENDSYNC.Time = End_Time;
            Broadcast(ENDSYNC);
            Wait until (SYSTEMCLOCK >= ENDSYNC.Time);
            Remove AIM MESSAGE from the AIM list;
            UpdateCAM();
        }
    }
}
```
Figure 5-14: EndSyncL algorithm - flowchart
5.6.1.4 The Interaction Management Engine (IME) Services and Functions

The interaction management engine (IME) contains three services (UpdateAPI, UpdatePIM, and SendPOM) and two functions (StartSync and UpdateCAM).

UpdatePIM Service (See Table 5-15 and Figure 5-15)

This service continuously checks for potential output messages in the PIM list for activation. If the LVC reaches the most imminent POMs (i.e., FIRSTPOM) timestamp, it calculates the scaling factor for the POM and calls the StartSync function by passing the activated POM. After the StartSync returns, it removes the activated POM from the PIM list.

If the PIM list is empty (i.e., no potential interactions in the DSS), this service goes into the sleep mode. In the sleep mode, a service stops its execution until it is woken up by another function or service in the LSA. The reason for using a sleep mode for services is to conserve processing power when there is no need for a service to run.

```c
service UpdatePIM() {
    While (Simulation is running)
    {
        If (PIM list is not empty)
        {
            FIRSTPOM = the POM with the minimum timestamp in the PIM list;
            If (LVC >= FIRSTPOM.Timestamp)
            {
                Calculate FIRSTPOM.Scaling_Factor;
                StartSync(FIRSTPOM);
                Remove FIRSTPOM from the PIM list;
            }
        }
        Else
        {
            FIRSTPOM = NULL;
            Sleep;
        }
    }
}
```
Figure 5-15: UpdatePIM algorithm - flowchart
UpdateCAM function (See Table 5-16 and Figure 5-16)

This function updates the CURRENT_ACTIVE_MESSAGE object and is called if a change has occurred in the PIM and AIM lists. It first checks if the AIM list is empty. If the list is not empty, it searches and finds the POM with the minimum scaling factor in the AIM list and sets it as the CURRENT_ACTIVE_MESSAGE. Also, updates the Current_Scaling_Factor to match the scaling factor of the CURRENT_ACTIVE_MESSAGE.

If the AIM list is empty, then it sets the execution mode to Regular_Execution (i.e. not paced), CURRENT_ACTIVE_MESSAGE to null and Current_Scaling_Factor to zero.

```java
function UpdateCAM()
{
  If (AIM list is not empty)
  {
    CURRENT_ACTIVE_MESSAGE = The POM with the minimum scaling factor in the AIM list;
    Current_Scaling_Factor = CURRENT_ACTIVE_MESSAGE.Scaling_Factor;
  }
  Else
  {
    MODE = Regular_Execution;
    CURRENT_ACTIVE_MESSAGE = NULL;
    Current_Scaling_Factor = 0;
  }
}
```
Figure 5-16: UpdateCAM algorithm – flowchart

**UpdateAPI service** (See Table 5-17 and Figure 5-17)

This service continuously iterates through the potential output messages in the PIM list and updates the allocation priority index and update frequency of messages.
```java
service UpdateAPI()
{
    While (Simulation is running)
    {
        For (all messages in the PIM list)
        {
            CURRENTPOM = currently addressed POM in the PIM list;
            Calculate CURRENTPOM.API;
            Calculate CURRENTPOM.UpdateFrequency;
        }
    }
}
```

Figure 5-17: UpdateAPI algorithm - flowchart
**StartSync function** (See Table 5-18 and Figure 5-18)

This function is called when a POM in the PIM list becomes active. First, the functions checks if the LSA is the leader for the activated potential output message (i.e. ACTPOM). If LSA is the leader, it waits until all acknowledgement messages (i.e. ACK) regarding the activated message are received from other LSAs. After all ACK messages are received, it created a new STARTSYNC message for the activated message and broadcasts it.

If LSA is not the leader for the activated POM, the function creates a new ACK message and sends it to the leader. After that, it waits until the STARTSYNC message from the leader is received. Once the STARTSYNC is received, it extracts the starting time from the STARTSYNC message and removes it from the input buffer.

The starting time indicates when LSAs are allowed to insert the activated message in the AIM list. It is set by the leader for synchronous starting of pacing. Once the starting time is reached, the function sets the execution mode to PacedExecution if the AIM list was empty (i.e. regular execution). It then appends the activated POM to the AIM list and calls UpdateCAM function to update the current active message.
function StartSync(ACTPOM)
{
    if (LSA is the LEADER for ACTPOM)
    {
        Wait until ((N-1) ACK messages are received);
        Remove ACK messages from the CM input buffer;
        STARTSYNC.ID = ACTPOM.ID;
        STARTSYNC.Time = SYSTEMCLOCK + Delay;
        Broadcast(STARTSYNC);
    }
    else
    {
        ACK.ID = ACTPOM.ID;
        Send(ACK, LEADER for ACTPOM);
        Wait until STARTSYNC message is received;
        Start_Time = STARTSYNC.Time;
        Remove STARTSYNC message from the CM input buffer;
    }
    Wait until (WALLCLOCK == Start_Time);
    if (AIM list is empty)
    {
        MODE = Paced_Execution;
    }
    Append ACTPOM to the AIM list;
    UpdateCAM();
}
Figure 5-18: **StartSync algorithm – flowchart**
SendPOM service (See Table 5-19 and Figure 5-19)

This service monitors the local POM list and acts upon a change in this list. It also broadcasts the updated POM information according to the update frequency determined by the allocation priority index (API).

There are two cases: a new potential output message is inserted in the POM list and a potential output message is removed from the POM list. In the first case, the service broadcasts the new POM, sets its status, initializes its `Last_Broadcast_Time` attribute and adds it to the PIM list. In the second case, the service broadcasts a CANCELPOM message with the information of the cancelled POM and removes the cancelled POM from the PIM list.

At each cycle, this service checks if an existing POM is updated and if it is, it broadcasts the updated POM according to the update frequency determined by the POMs allocation priority index. For this, the service checks the amount of time elapsed since the last time this POM was broadcasted. If the amount of elapsed time is greater than the minimum allowed time between two broadcasts (reciprocal of the update frequency) then the POM is broadcasted and its status is set back to “Current”.
Table 5-19: SendPOM algorithm – code

```java
service SendPOM()
{
    While (simulation is running)
    {
        If (a new POM entered the POM list)
        {
            NEWPOM = the new POM in the POM list;
            Broadcast(NEWPOM);
            NEWPOM.Status = Current;
            NEWPOM.Last_Broadcast_Time = SYSTEMCLOCK;
            Append NEWPOM to the PIM list;
        }
        If (a message is removed from the POM list)
        {
            OLDPOM = the message removed from the POM list;
            CANCELPM.ID = OLDPOM.ID;
            Broadcast(CANCELPM);
            Remove OLDPOM from the PIM list;
        }
        For (all messages in the POM list)
        {
            CURRENTPOM = the POM currently addressed in the POM list;
            If (CURRENTPOM.Status == Updated)
            {
                Elapsed_Since_Last_Broadcast = SYSTEMCLOCK -
                CURRENTPOM.Last_Broadcast_Time;
                If ((1/CURRENTPOM.Update_Frequency) <=
                Elapsed_Since_Last_Broadcast)
                {
                    Broadcast(CURRENTPOM);
                    CURRENTPOM.Status = Current;
                    CURRENTPOM.Last_Broadcast_Time = SYSTEMCLOCK;
                }
            }
        }
    }
}
```
Figure 5-19: SendPOM algorithm - flowchart
5.6.2 The Core Synchronization Service Algorithm

The core synchronization service (CSS) is composed of a Main procedure that loops continuously during runtime and four sub-procedures that are called from the Main procedure.

CSS runs in LSA shell and is the major algorithm that binds all other functions and services together to form the synchronization system. It is generic and the same for all LSAs regardless of the characteristics of the underlying simulation models. Therefore, the algorithm need not be modified for specific simulation models.

Main procedure (See Table 5-20 and Figure 5-20)

The Main procedure loops continuously until the simulation is completed. It first calls the function Initialize to initialize the data structures and other functions used in the algorithm before the runtime. After initialization, it moves into a continuous loop where it cycles between regular and paced execution by calling functions that check whether conditions in the DSS/LSA require regular or paced execution. After the simulation end time is reached by the local virtual clock (LVC), it calls the Terminate function to finalize and quit the simulation synchronization.

```c
service MAIN()
{
    Initialize();
    While (Simulation is running)
    {
        UnpacedExecute();
        PacedExecute();
    }
    Terminate();
}
```
**Initialize function** (See Table 5-21 and Figure 5-21)

The **Initialize** function first initializes POM, PIM and AIM lists before the execution starts. If there are entities that are scheduled to be created at virtual time zero, these entities are registered with the IME by creating structural object models and the potential messages related to them are added to the initial POM list.

Once the POM list is populated by potential messages, entire contents of this initial POM list is broadcasted. After the broadcast is complete, a DONE message is broadcasted. This message signals the end of the broadcast of POMs and also carries a checksum so that receiver LSAs can be sure of complete receipt of all broadcasted POM information.
After the DONE notice is broadcasted, the procedure waits until all POMs sent by other LSAs are received. This is done by checking DONE notices received from other LSAs against the POMs received. Once this check is complete and all broadcasted POMs are received by the LSA, received POMs are inserted into the PIM list.

After the PIM list is populated by the received POMs, the POM with the smallest timestamp is identified and if the timestamp of this POM is zero, the execution mode is set to \textit{PacedExecution}, otherwise it is set to \textit{RegularExecution}.

After the \texttt{MODE} variable is set, a READY notice is broadcasted to signal that the LSA is ready to start execution. After this broadcast, the function waits for all READY messages to be received and once this is complete, the procedure terminates and program pointer passes back to the \texttt{MAIN} procedure.
function Initialize()
{
    Initialize POM, PIM and AIM lists;
    For (all entities scheduled to be created at LVC=0)
    {
        NEW_ENTITY = currently addressed new entity;
        Create new entity-specific structural model object of the type-
        NEW_ENTITY.Type;
        For (all messages in the initial state-table of NEW_ENTITY)
        {
            MESSAGE = currently addressed message in the initial state-
            table of NEW_ENTITY;
            MESSAGE.Status = Current;
            Append MESSAGE to the POM list;
        }
    }
    Broadcast(POM list information);
    Broadcast(DONE);
    While ((Number of DONE notices received < N-1) AND (All N-1 -
        Checksums are not OK))
    {
        Receive incoming POMs;
    }
    Append received POMs to the PIM list;
    PIM_MESSAGE = the POM with the smallest timestamp in the PIM list;
    If (PIM_MESSAGE.Timestamp == 0)
    {
        MODE = PacedExecution;
    }
    Else
    {
        MODE = RegularExecution;
    }
    Broadcast(READY);
    Wait until (All READY notices received from LSAs);
Figure 5-21: Initialize algorithm – flowchart
**RegularExecute function** (See Table 5-22 and Figure 5-22)

The **RegularExecute** function is one of the two main functions that the **MAIN** procedure cycles between. As the name implies, it is used when no interaction is in effect in the DSS. In this case, the simulation process advances with no pacing.

This function first checks the simulation execution mode. The **MODE** variable is set initially by the **Initialize** function and later by the **UpdateCAM** and **StartSync** functions. This variable identifies whether pacing is required. Since, this is the first function visited when the simulation starts, if the simulation needs to enter the paced execution mode at virtual time zero, this function terminates without executing itself.

If pacing is not required initially, then the function proceeds by checking the predicted firing time of the most imminent potential output message (POM) in the PIM list. If the timestamp of the next event in the event calendar of the simulation process is less than the timestamp of the closest interaction, then it is safe to execute the event.

After the event is executed, the variable holding the most imminent POM (i.e. **PIM_MESSAGE**) is re-sampled and the condition to execute the next event in the calendar is checked. The procedure loops in this cycle until the next event in the calendar is not safe to execute. After this condition is reached, the LVC is advanced to the timestamp of the **PIM_MESSAGE** and the function terminates.
function RegularExecute()
{
    if (MODE == Regular_Execution)
    {
        PIM_MESSAGE = the message with the smallest timestamp in the PIM list;
        while (Nextevent.Timestamp < PIM_MESSAGE.Timestamp)
        {
            Execute Nextevent;
            PIM_MESSAGE = the message with the smallest timestamp in the PIM list;
        }
        LVC = PIM_MESSAGE.Timestamp;
    }
}
**PacedExecute function** (See Table 5-23 and Figure 5-23)

The PacedExecute function first waits until the MODE is switched to Paced_Execution because the stand-alone running functions responsible for updating the MODE variable may not execute as quickly as the MAIN procedure switches from RegularExecute function to PacedExecute function.
After the MODE is verified, StartPacing function is called. Once the pacing is started, the function executes the next event in the event calendar of the simulation process if the timestamp of the event is less than the TIMER (see StartPacing function for an explanation of TIMER functioning) value. This loop continues until the MODE changes back to Regular_Execution which means there are no active interactions left that requires pacing. Once this condition is reached, the pacing is stopped by calling the StopPacing function and the function terminates.

```
function PacedExecute()
{
    Wait until (MODE == Paced_Execution);
    StartPacing();
    While (MODE == Paced_Execution)
    {
        If (NextEvent.TimeStamp <= TIMER)
        {
            Execute NextEvent;
        }
    }
    StopPacing();
}
```
Terminate function (See Table 5-24 and Figure 5-24)

The Terminate function first checks if the simulation is in paced execution mode and if this is the condition then it calls the StopPacing function to stop the pacing activities and then terminates.

```c
function Terminate()
{
    if (MODE == PacedExecution)
    {
        StopPacing();
    }
}
```
5.7 APP Synchronization Framework – Concluding Remarks

The adaptive partial pacing (APP) algorithm is a collection of several individual algorithms. These individual algorithms are categorized into functional components which come together to form the local synchronization agent. Algorithms are also classified as services and functions. Services run continuously during an execution while functions run only when they are called from within a services or another function.

APP algorithm is designed in two parts: (1) a core synchronization service and (2) auxiliary services and functions. The core synchronization service starts running before the simulation execution begins and terminates after the simulation execution ends. It is the main program that maintains the synchronization in the DSS. Auxiliary services and functions on the other hand, help the core service maintain the data structures, execute communication requests, provide timing services and maintain proper system-clock synchronization.
APP algorithm was designed with the assumption that communication is reliable. Every message sent out from an LSA/LP should be guaranteed to be received by the destination LSA/LP within a predefined maximum amount of time — maximum transmission time. If the communication network used cannot guarantee this level of reliability, APP algorithm cannot guarantee synchronization.

The overall APP synchronization framework was designed to be suitable for automated synthesis. As mentioned in the algorithm sections, all algorithms are generic and do not depend on simulation model specifics. Automatic generation first starts with generating structural model classes (i.e., templates) by analyzing simulation models to be run in parallel. This subject was discussed in detail in Chapter 4. After structural model generation is completed, the LSA is generated by inserting the structural model core (SMC) into the pre-defined LSA implementation and the system will be ready to run.

Certain services and functions of the LSA depend on the specifics of the simulation software and the communication medium used in the DSS. Therefore, these services and functions must be implemented specifically for the simulation software and the communication medium being used for distributed simulation. However, these services only need to be implemented once after the simulation software and communication medium is selected and does not affect the scalability and portability of the framework in terms of different simulation models created by the same simulation software.
Chapter 6
Performance Analyses

This chapter examines the performance characteristics of the Adaptive Partial Pacing (APP) synchronization framework and its underlying theory. The APP algorithm was presented in detail in Chapter 5. The mathematical analyses of the pacing requirements were provided in Chapter 3.

The chapter is organized as follows: Section 6.1 provides a basic proof of correctness for the APP algorithm (validation). In Section 6.2, experimental analyses of the speed-up performance of the APP framework against sequential (single-processor) simulation are presented. Finally, Section 6.3 discusses the overall performance characteristics of the APP framework in the light of the analyses provided in this chapter.

6.1 Proof of Correctness of the APP Synchronization Algorithm

The Adaptive Partial Pacing (APP) synchronization algorithm depends on time-stepped pacing of simulation executions for distributed simulation processes (LPs) during potential interactions in a distributed simulation system (DSS). The APP algorithm is divided into two main phases: (1) regular execution and (2) paced execution.

Regular execution means unrestricted event-driven execution of a simulation process. This is the case in sequential simulations. The APP algorithm stays in the regular execution phase unless there exists at least one imminent potential output message in the DSS. Thus, in the simplest case where simulation processes do not interact at all, all local synchronization agents (LSAs) always stay in the regular execution phase and simulation executions take place independently in parallel in an event-driven (as fast as possible) fashion.
On the other hand, the APP algorithm switches to the paced execution phase when an imminent message exists in the potential interaction messages (PIM) list. Imminence of a potential interaction message is measured by the amount of virtual time between the current local virtual clock and the earliest possible virtual time of firing of the potential interaction message. Therefore, as long as potential output messages (POMs) are detected before they fire, placed in the local PIM list and communicated to other LSAs to be placed in remote PIM lists, all LSAs are aware of a potential interaction message before it actually fires, and they switch to paced execution phase.

The requirements for conservative synchronization of parallel discrete event simulations were discussed in detail in Chapters 2 and 3. A sufficient condition for synchronization is coordinated execution (i.e. difference between the local virtual clocks of interacting simulations is less than the amount of minimum virtual interaction lag) of simulation processes during interactions. In Chapter 3, the lower bound on the time-scaling factor for synchronous execution is provided along with the proof. Therefore, once the execution of simulation processes switch to paced execution phase before the interaction message actually fires, it is guaranteed (due to the pacing and accompanying assumptions as declared in Chapter 3) that the synchronization will be protected until the interaction message is received by the target simulation process.

In the case of multiple overlapping interaction messages, the APP algorithm chooses the maximum time-scaling factor among the active messages to pace the distributed simulation processes. Therefore, it is still guaranteed that all active potential interactions will take place without losing synchronization. However, one must note that an important assumption of the pacing based synchronization is that the virtual interaction lag of a message is positive. If any interaction between two physical processes is modeled as an instantaneous delay in the simulation, the APP algorithm cannot be used to preserve synchronization in such a DSS.
Consequently, if all messages are detected and propagated across the DSS before they actually fire, the APP algorithm guarantees preservation of the synchronization state in the DSS. Thus, it only remains to prove that the DSS execution starts at a synchronized state and messages are detected and propagated before they fire during an execution.

Considering the assumption (presented in Chapter 3) that the virtual time spent at any non-dummy state is greater than zero, the proof of the first argument is straightforward. Since all simulation processes start with empty event calendars, no event exists in the DSS and therefore at virtual time zero all simulation processes are automatically synchronized. However, even if the no zero state-delay assumption is relaxed, the APP algorithm initializes the POM, PIM and AIM lists before the execution starts and checks for any output events that are scheduled to fire at virtual time zero. Therefore, if this is the case, the system directly starts in paced execution phase.

As for the timely detection and propagation of interaction messages, the detection and propagation mechanisms work as follows: the simulation control engine (SCE) updates the state of all active entities in the structural model core after every event execution during regular execution phase. When an event schedules the event for creation of a new entity, all potential messages related to the new entity are automatically inserted into the local POM and PIM lists. In between executions, the interaction management engine (IME) broadcasts all new potential messages to other LSAs.

Note that, each entity can fire at most one output event and an output event can be fired by only one entity. Since an LSA already knows the types of entities that are capable of firing output events (due to entity specific structural models), it is only a matter of keeping track of these output event capable entity types and keeping the PIM list always non-empty (i.e. always knowing the virtual time of creation of the next entity and inserting its potential output events into the PIM list before the entity is created).
At virtual time zero (before the simulation execution starts), by using the minimum interarrival time (or the initial creation time of an entity), an LSA populates the PIM list with the potential messages of the first entities corresponding to all output event capable entity types and propagates them to all remote LSAs. Since the next entity of a certain type will be created after the creation of the first entity, any of the potential messages of the next entity cannot fire earlier than the earliest possible firing time of the potential messages of the first entity. And since at the time of creation of the first entity the creation event of the next entity is scheduled, once the first entity is created the next entity’s potential events are also inserted into the PIM list and broadcasted. Therefore, by induction any consecutive entity’s potential messages will be detected and propagated on time before they actually fire. This completes the proof of correctness of the APP algorithm.

6.2 Performance Analyses of the APP Framework

Among many objectives of parallel simulation, one objective is to shorten the amount of time it takes to complete the simulation execution, henceforth referred to as “speed-up”. Although speed-up is not the main objective of the system developed in this research, an analysis of the speed-up characteristics of the APP framework is provided in this chapter to better understand the factors affecting the performance of the developed synchronization system.

As stated earlier in Chapter 3, the speed-up gained by parallelizing a sequential simulation is very much dependent on the performance of the synchronization mechanism. The synchronization operation introduces an inevitable overhead into the simulation process that is mostly attributable to communication between simulation processes during parallel execution.

The performance of the APP synchronization algorithm depends mostly on the margin of error made by predicting the firing time of a potential interaction in the DSS.
The APP algorithm makes use of minimum state delays and shortest path calculations to obtain earliest virtual time of firing of output events. Since this is a crude estimate of the firing time, the firing time estimate is updated as frequently as possible (after every state transition) during execution. Therefore, the margin of error in predicting the firing time depends on the cumulative variability in the length of delays at the states visited by an entity on the way to an output event.

Apart from the variability issue, another important factor that determines the performance of the framework is the “degree of coupling” between interacting simulation processes in the DSS. Intuitively, it is reasonable to state that as the number of interactions taking place in unit time increases, the APP algorithm would spend more time in paced execution phase and hence the overall completion time will be longer. Therefore, the degree of coupling will have a profound effect on the performance of the APP framework (as for any other synchronization framework) as a tool for parallel execution of simulation processes.

In this section, the effect of these two factors, namely (1) variance of state delays and (2) degree of coupling between interaction simulation processes, on the performance of the APP framework is experimentally analyzed.

### 6.2.1 Experimental Setup

The experiments reported in this chapter are conducted on a two computer distributed simulation system connected via a wireless local area network. The distributed system specifications are provided in Table 6-1.
Local synchronization agents and the APP algorithm are developed using Microsoft .NET framework v1.1 (programmed in Visual C# language) and are compatible with the Microsoft Common Language Runtime (CLR) so that they can be ported to other operating systems in the future without substantial modification to the code-base.

The interpreter (piece of software that converts an Arena™ model into general and entity specific structural models as discussed in detail in Chapters 3 and 4) was developed using Microsoft Visual C++ v6.0.

During these experiments, only one dedicated local area network is used for both synchronization and simulation interaction purposes.

The parameters used in the calculation of the time-scaling factor are selected as listed in Table 6-2 after initial testing of the communication network round-trip times and computer clock synchronization accuracy.

Table 6-1: Specifications of the Distributed Simulation System Components

<table>
<thead>
<tr>
<th>Computers</th>
<th>Processor</th>
<th>RAM</th>
<th>Storage</th>
<th>Network Card</th>
<th>Operating System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer A</td>
<td>Intel Pentium III (1.10 GHz)</td>
<td>512 Mb DDR SDRAM</td>
<td>30 Gb</td>
<td>Linksys 802.11b wireless networking card (11 Mb/s)</td>
<td>Microsoft Windows XP Professional</td>
</tr>
<tr>
<td>Computer B</td>
<td>AMD Athlon XP-M 2000+ (1.667 GHz)</td>
<td>512 Mb</td>
<td>40 Gb</td>
<td>Built-in 802.11g wireless networking card (54 Mb/s)</td>
<td>Microsoft Windows XP Home Edition</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Network Router</th>
<th>Network Standard</th>
<th>Network Bandwidth</th>
<th>Network Protocol</th>
<th>Encryption</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linksys Wireless-B Broadband Router</td>
<td>IEEE 802.11b</td>
<td>Up to 11 Mb/s</td>
<td>TCP/IP</td>
<td>128-bit WEP</td>
</tr>
</tbody>
</table>
Table 6-2: Values of the parameters used in time-scaling factor calculations

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Parameter Symbol</th>
<th>Parameter Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum communication delay (*)</td>
<td>Δt_{MAX}</td>
<td>50 milliseconds</td>
</tr>
<tr>
<td>Maximum clock synchronization error</td>
<td>τ_{sync}</td>
<td>100 milliseconds</td>
</tr>
<tr>
<td>Reference Clock Frequency</td>
<td>ρ_{ref}</td>
<td>1.0</td>
</tr>
<tr>
<td>Maximum frequency drift rate</td>
<td>ρ_{f_drift}</td>
<td>0.000001 s^{-1}</td>
</tr>
<tr>
<td>Clock synchronization interval</td>
<td>t_{sync}</td>
<td>500 milliseconds</td>
</tr>
</tbody>
</table>

(*)one-way, including processing time at originating and target computers and the network router

The example physical system depicted in Figure 4.2 of Chapter 4 is slightly modified and used for the analyses. There are two entity types: type A and type B. In simulation process 1 (LP1) (structural models of LP1 and LP2 are depicted in Figure 6-1 and Figure 6-2 respectively), at state χ_{10} entities of type A are transported to LP2 (output message m_{trans}^{(7,10)}). Transferred entities of type A arrive in LP2 at state χ_{1} (input message m_{enter}^{(0,1)}). Entities of type B are not transported. In implementation, these two messages correspond to one interaction message that couples these two simulation processes.

In both simulation processes LP1 and LP2, at each state, all possible transition events have equal probability of being selected as the next event. For example, at state χ_{1}, possible events σ_{process}^{(1,4)} , σ_{process}^{(1,5)} and σ_{process}^{(1,6)}, each has approximately 33% chance of being selected as the next transition.

As for the state delays, every state has a random delay distributed uniformly between a lower-bound and an upper-bound. The ranges of these state delay random distributions are varied to analyze the effect of variability of state delays on the performance of the APP algorithm.
Figure 6-1: General structural model of LP₁ – states marked with enabled entity types

Figure 6-2: General structural model of LP₂ – states marked with enabled entity types
6.2.2 Effect of State-Delay Variability on APP Algorithm Performance

Variability of a state-delay is measured by its range, that is upper-bound (maximum delay time - \( u_i \)) less the lower-bound (minimum delay time - \( l_i \)) of the corresponding uniform distribution (Eq. 6.1).

\[
\text{var}(\chi_i) = u_i - l_i, \quad \forall \chi_i \in X
\]  

(6.1)

A single variability measure for an entity-specific structural model is formed by adding individual variability measures (ranges) of all states in a path to a potential output message (Eq. 6.2).

\[
\text{var}(S_{\chi_i}) = \sum_{\forall X_i \in P} \text{var}(\chi_i), \quad \text{such that:}
\]

\[
P = \{ \chi_k : X_k \rightarrow X_m, \chi_m \xrightarrow{e_n} X_{m+1}, e_m \in S_{\chi_i}, \Sigma_{\text{Output}} \text{ and } \chi_k, \chi_m, X_{m+1} \in \Xi \chi_i \}
\]  

(6.2)

The performance of the algorithm is measured in terms of “speed-up” with respect to the sequential simulation of the combination of these two simulation processes. Speed-up is measured by calculating the ratio of the completion time of parallel simulation to that of the sequential simulation and subtracting this ratio from one. The combined simulation process is simply formed by replacing the matching output and input messages with an internal event and changing the type of events and states accordingly (see Chapter 3 for the formal definition of the composition operation).

Each experiment is run once in parallel (2 simulation processes) and once in sequential (single combined simulation process) using common random numbers (i.e. same pseudo random number seed). The simulation processes are intentionally slowed down (a constant real-time delay, e.g. 10 milliseconds, is inserted into each event execution) to create an efficiency gap between parallel simulation and sequential simulation. Slowing down execution of one event creates the effect of executing several hundreds of events in stead of a single event and therefore makes parallel execution faster than sequential without creating very large simulation models with thousands of nodes. This issue is explained in detail in the next subsection.
For the same reason, the simulation time between arrivals of consecutive entities of type A in LP\textsubscript{1} is significantly larger compared to the other delay amounts. The simulations are run for 40,000 virtual time units which correspond to approximately four potential interactions.

The experimental design for this analysis is presented in Table 6-3 with the corresponding lower and upper bound values for state delays. The values in this table correspond to the simulation process 1 (LP\textsubscript{1}) since it is the only process sending output messages (values for state $\chi$ are zero since it is dummy state). The values of the simulation process 2 (LP\textsubscript{2}) are held constant throughout the experiments as well as the virtual interaction lag.

The results of the experiments are shown in Table 6-4 and charted in Figure 6-3. As seen from these results, increasing variability of state-delays cause speed-up percentage to decrease. This is due to the APP algorithm spending more simulation time in the paced execution phase in which the execution is substantially slower compared to regular execution phase resulting in increased coordination overhead.
Table 6-3: Experimental design for variability of state-delay analysis (LP$_1$)

<table>
<thead>
<tr>
<th>Run no.</th>
<th>$\lambda_i$</th>
<th>$\lambda_1$</th>
<th>$\lambda_4$</th>
<th>$\lambda_5$</th>
<th>$\lambda_6$</th>
<th>$\lambda_7$</th>
<th>var($\lambda_i$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10020</td>
<td>0</td>
<td>25</td>
<td>15</td>
<td>35</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>10020</td>
<td>0</td>
<td>25</td>
<td>15</td>
<td>35</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>10016</td>
<td>0</td>
<td>20</td>
<td>12</td>
<td>25</td>
<td>8</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>10020</td>
<td>0</td>
<td>25</td>
<td>15</td>
<td>35</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>10012</td>
<td>0</td>
<td>15</td>
<td>9</td>
<td>21</td>
<td>6</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>10020</td>
<td>0</td>
<td>25</td>
<td>15</td>
<td>35</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>10008</td>
<td>0</td>
<td>10</td>
<td>6</td>
<td>14</td>
<td>4</td>
<td>63</td>
</tr>
<tr>
<td></td>
<td>10020</td>
<td>0</td>
<td>25</td>
<td>15</td>
<td>35</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>10004</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>7</td>
<td>2</td>
<td>84</td>
</tr>
<tr>
<td></td>
<td>10020</td>
<td>0</td>
<td>25</td>
<td>15</td>
<td>35</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

Table 6-4: Results of the experiments for variability of state-delay analysis

<table>
<thead>
<tr>
<th>Run No</th>
<th>Parallel Runtime (ms)</th>
<th>Sequential Runtime (ms)</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>86824</td>
<td>127373</td>
<td>31.83%</td>
</tr>
<tr>
<td>2</td>
<td>94846</td>
<td>135645</td>
<td>30.96%</td>
</tr>
<tr>
<td>3</td>
<td>95988</td>
<td>135344</td>
<td>29.08%</td>
</tr>
<tr>
<td>4</td>
<td>97880</td>
<td>135334</td>
<td>27.67%</td>
</tr>
<tr>
<td>5</td>
<td>124569</td>
<td>164867</td>
<td>24.44%</td>
</tr>
</tbody>
</table>
6.2.3 Effect of Degree-of-Coupling on APP Algorithm Performance

The “degree of coupling” is a measure of intensity of interactions within a DSS and following are two possible measures of the degree of coupling:

1. Average number of interactions taking place per unit time.
2. Processing load of the internal events of distributed simulation processes.

The first measure is a direct indicator of the intensity of actual interactions taking place in a DSS during parallel execution. This measure depends on the number of output/input messages between simulation processes and the rate of firing of these messages. In other words, it can be calculated as the rate of transportation of entities between simulation processes.

The rate of transportation of entities is equivalent to the rate of departure of entities from output event firing states. Therefore, by keeping statistics on departure processes one can calculate the intensity of interactions in a DSS. Note that one needs to run the DSS to obtain these statistics.
On the other hand, it is possible to obtain an upper-bound on the total number of interactions during an execution by calculating the maximum number of message-producing entities that can be created during an execution. The maximum number of entities created during an execution can be calculated by dividing the total execution duration by the minimum interarrival time for an entity type (lower-bound of the delay at an initial state). Since each entity can produce at most one interaction, the maximum number of entities must be equal to the maximum number of interactions possible during a predefined duration of simulation execution.

The second measure is a more important indicator of coupling between simulation processes, but not in an actual sense like the first measure. When a sequential simulation process is partitioned into several smaller simulation processes for parallel execution, for each of these smaller simulation processes, if the processing load of interactions are large compared to the processing load of internal events, the amount of extra processing time spent for synchronization at each interaction would surpass the benefit of partitioning the sequential simulation process.

The above mentioned situation creates a virtual coupling between partitioned simulation processes where it would be more efficient to combine these small simulation processes to form larger partitions in which the processing load of internal events is substantial.

In this respect, the second measure can be estimated by examining the number of states an interaction-producing entity visits between its creation and its transportation (i.e. firing of an interaction message). This number is directly proportional to the number of internal events on the path of that entity and as the number of events executed increases the amount of time required to process these events would also increase.
Consequently, one can estimate the second measure by simply multiplying the minimum number of events that must be executed for an entity to reach an output event with the average amount of processing time required to execute a single event.

Intuitively, if the second measure is small, then sequential simulation should be faster than parallel simulation and in this case parallel execution efficiency decreases due to tight coupling of simulation processes. On the other hand, as this factor gets larger, the speed-up due to parallelization would also improve up to a certain point where the limit of speed-up by parallel execution is reached.

In order to demonstrate the effect of above given two factors, two sets of experiments (one for each measure) were conducted. The parameters used for experiment set 1 are presented in Table 6-5 and for experiment set 2 in Table 6-7.

In these experiments, factor 1 is varied by changing the minimum interarrival time of the interaction-producing entity type that is entity type A. Factor 2 is varied by inserting variable amounts of artificial processing delays in execution of internal events. This method is more convenient than increasing the number of states in the simulation process and yields the same effect of increasing the processing load of internal events in the simulation process.

The results of the first set of experiments are provided in Table 6-6 and charted in Figure 6-4. The results show that, as the maximum number of interactions increase, the performance of overall parallel execution decreases significantly due to the increasing coupling between simulation processes. For this particular system, beyond five maximum interactions per 40,000 virtual time units of simulation execution, parallel execution time is greater than sequential execution time.

Unfortunately, this threshold will be different for different simulation processes and it is not possible to test all possible simulation scenarios to come up with a general
threshold value to map out the performance characteristics of parallel execution using
APP framework. However, one can make test runs to determine this threshold for
different partitioning schemes of a distributed simulation system and select the
partitioning scheme with the best results for further use. This issue is not discussed here
in detail, since efficient partitioning of a single simulation process into multiple
distributed simulation processes is beyond the scope of this research.

Table 6-5: Experimental design for factor 1 (LP₁)

<table>
<thead>
<tr>
<th>Run no.</th>
<th>( \chi_0 )</th>
<th>Maximum Number of Interactions (Simulation Duration = 40,000 vtu*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( l_i ) 1000</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>( u_i ) 1010</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>( l_i ) 3000</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>( u_i ) 3010</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>( l_i ) 5000</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>( u_i ) 5010</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>( l_i ) 7000</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>( u_i ) 7010</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>( l_i ) 9000</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>( u_i ) 9010</td>
<td></td>
</tr>
</tbody>
</table>

(*) vtu: Virtual time unit

Table 6-6: Results of the experiment for degree-of-coupling analysis – factor 1

<table>
<thead>
<tr>
<th>Run No</th>
<th>Parallel Runtime (ms)</th>
<th>Sequential Runtime (ms)</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>618218</td>
<td>126792</td>
<td>-387.58%</td>
</tr>
<tr>
<td>2</td>
<td>236610</td>
<td>128224</td>
<td>-84.53%</td>
</tr>
<tr>
<td>3</td>
<td>187960</td>
<td>126692</td>
<td>-48.36%</td>
</tr>
<tr>
<td>4</td>
<td>123647</td>
<td>126341</td>
<td>2.13%</td>
</tr>
<tr>
<td>5</td>
<td>104633</td>
<td>127082</td>
<td>17.66%</td>
</tr>
</tbody>
</table>
In the second set of experiments, average event processing time is calculated by dividing the total runtime by the total number of events. Total number of events for 40,000 virtual time units was determined as 12,609.

The results of the second set of experiments are provided in Table 6-8 and charted in Figure 6-5.

Table 6-7: Experimental design for factor 2

<table>
<thead>
<tr>
<th>Run No.</th>
<th>Avg. Event Processing Time (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>23</td>
</tr>
<tr>
<td>2</td>
<td>307</td>
</tr>
<tr>
<td>3</td>
<td>1062</td>
</tr>
<tr>
<td>4</td>
<td>11682</td>
</tr>
<tr>
<td>5</td>
<td>29118</td>
</tr>
</tbody>
</table>
Table 6-8: Results of the degree-of-coupling experiments - factor 2

<table>
<thead>
<tr>
<th>Run No</th>
<th>Parallel Runtime (ms)</th>
<th>Sequential Runtime (ms)</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4236</td>
<td>290</td>
<td>-1360.69%</td>
</tr>
<tr>
<td>2</td>
<td>6949</td>
<td>3875</td>
<td>-79.33%</td>
</tr>
<tr>
<td>3</td>
<td>13714</td>
<td>13149</td>
<td>-4.30%</td>
</tr>
<tr>
<td>4</td>
<td>106454</td>
<td>147532</td>
<td>27.87%</td>
</tr>
<tr>
<td>5</td>
<td>212736</td>
<td>367728</td>
<td>42.15%</td>
</tr>
</tbody>
</table>

Figure 6-5: Results of the experiments for degree-of-coupling analysis – factor2

The results of the second set of experiments suggest that as the processing load of internal events increase, the virtual degree of coupling between simulation processes decrease and parallel execution becomes more and more efficient. This is a clear indication of the fact that gain from parallel execution exceeds the synchronization overhead only after the simulation process becomes substantially large.

As discussed in the light of the previous experiment results, it is not possible to calculate a general threshold value of measure 2 for all possible simulation processes. The break even point should be determined by conducting test runs.
6.3 Concluding Remarks on Performance Characteristics of the APP Synchronization Framework

In this chapter, first a proof of correctness for the APP synchronization algorithm is presented, followed by the experimental analyses of performance of the APP framework. Two important factors: (1) variability of state-delays and (2) degree of coupling between simulation processes, were showed to have significant effect on the performance of the APP synchronization framework.

Like any other existing synchronization mechanism, pacing based synchronization is not free of synchronization overhead. Variability associated with the amount of delay an entity incurs at states has a cumulative effect on the amount of error made in predicting the firing time of a potential interaction message. Prediction errors cause the APP algorithm to stay in paced execution phase longer than necessary, resulting in increased overhead.

Although the APP algorithm updates these timing predictions as network bandwidth is available, it is not possible to completely eliminate the prediction error for stochastic systems. Therefore, one can conclude that the APP framework will be more suitable for distributed simulation of low variability systems, such as manufacturing systems where most of the processes are automated.

In this respect, large-scale semi-conductor and electronics manufacturing operations are good candidates for utilization of the APP framework based distributed simulation. At the far extreme, any system with deterministic delays and routings would yield the maximum performance of the APP framework in terms of the variability factor. Although simulations of deterministic systems are rare in the manufacturing industry, systems like shared-memory parallel computers have deterministic behavior by their nature and thus are very good candidates to be simulated in parallel using the APP framework.
Unlike the variability factor, the degree of coupling is harder to measure/estimate. Degree of coupling indicates the interdependence of simulation processes and is a key factor, as showed by the results of the experiments, in determining the efficiency of using distributed execution as opposed to sequential execution. This is in general true for not only the APP framework but for all synchronization mechanisms. As the degree of coupling between distributed processes increases, it becomes more costly to synchronize these processes. Even if synchronization overhead was zero (or negligible), increased degree of coupling means more communication between distributed processors (e.g. computers) and with the current networking technology (and probably for many future technologies to come) communication is several degrees of magnitude slower than computer processing. As expected, it is not very efficient to execute tightly-coupled simulation processes in parallel.

Two measures that estimate the degree of coupling between simulation processes are introduced in this chapter. These measures are not complete descriptions of the degree of coupling between distributed interacting processes, however they still provide a valuable insight to the coupling issue and the results of the experiments suggest that they are fairly good indicators of coupling.

The degree-of-coupling analyses clearly show the effect of coupling on the performance of the APP framework. It appears that the best way to evaluate the efficiency of a distributed simulation system is to actually test it.

In this respect, the APP framework has a solid advantage over existing synchronization mechanisms since it is substantially easier to develop and deploy distributed simulation systems using the APP framework once the necessary interpreters are developed for the utilized simulation software. This issue is discussed in more detail in Chapter 7 where the merits of the APP framework are discussed in comparison with the existing prominent synchronization mechanisms.
Chapter 7

Conclusions and Future Work

This chapter discusses the essential findings of this research and its contributions to the state-of-the-art in the light of the research objectives. It also provides ideas for future research.

In Section 1, a summary of the essential findings of the research is provided. Section 2 outlines the contributions of this research to the area of parallel and distributed simulation. In Section 3, areas of improvement in the developed synchronization framework and directions for future research are discussed.

7.1 Summary of Research Findings

The future of computer simulation promises to include modeling and analyses of very large scale and high-fidelity real-life systems. Businesses depend more and more on computerized models to predict outcomes of future events and decisions on systems ranging from social structures to physical phenomenon. Rough approximations and abstractions are not as desirable as they once were, due to the increasing complexity of products/services.

As computing power becomes more available and cheaper, it is possible to create clusters of cheap computers that collectively match the computing power of expensive super-computers. Parallel and distributed computing is a very important avenue to provide businesses with tools and technologies to satisfy the large scale and high-fidelity simulation needs. Development of efficient and effective distributed simulation tools and technologies will revolutionize the way analyses are conducted in many businesses. This revolution has already begun in academic and military areas and companies such as IBM
and Sun Microsystems are currently in the process of developing distributed computing tools for business enterprises.

This research resides in a small but important part of the vast area of parallel and distributed computing: distributed simulation of discrete event systems. Within this context, two fundamental problems in the path to develop efficient and effective distributed simulation systems were addressed:

- Synchronization (time management) of distributed simulations.
- Fast development and deployment of distributed simulation systems.

Synchronization is a prerequisite for a distributed simulation system to function properly (i.e., provide repeatable and accurate output) and has been researched extensively over the years. The amount of processing and communication overhead produced by a synchronization mechanism ultimately determines the run time efficiency of a distributed simulation system. The main objective of distributing a sequential simulation is to shorten the development and programming time required to create complex models and then run them. It is imperative that the synchronization requirements and overhead do not outweigh the gains made due to parallelization. That is, models that are decomposed into smaller pieces are much easier to create, validate, verify and assess. On the other hand, parallelizing these models creates the requirement for time coordination. The time and overhead associated with this parallelization should not outweigh gains made from parallelization.

The comparisons here include the tradeoff associated with modeling time and simulation time. Typically, modeling time will be orders of magnitude greater than simulation run time. The advantages obtained from model decomposition will far outweigh any time disadvantages lost from coordination overhead. It is almost impossible to measure the gains obtained from general decomposition; therefore, these will go unaddressed any further in this chapter. However, it is safe to say that modeling time gains for decomposition will be orders of magnitude different than run time gains or
losses. Since there are no disadvantages associated with modeling decomposition, it will not be addressed further in this chapter. The focus of the remainder of the chapter will be on run time tradeoffs.

In order to address the synchronization problem, a new synchronization mechanism based on coordinated pacing of distributed simulations was developed. This mechanism, dubbed Adaptive Partial Pacing (APP), detects interactions in a distributed simulation system (DSS) before they take place and coordinates a synchronous execution among all simulation processes until interactions are completed. More specifically, simulation processes run as fast as possible (no pacing) when there are no imminent interactions in the near future of the DSS and switch to paced execution when an interaction is about to start and stay in paced execution until the interaction is completed.

Paced execution is another name for time-stepped execution of a simulation process where the amount of wallclock time spent between to time-steps is equivalent to the pace of the execution with respect to real-time. The system-clocks of computers are used for pacing and since coordination is required, system-clocks are synchronized using an external reference clock.

Pace of execution is dynamically determined based on the characteristics of the DSS and interaction messages. Adaptive pacing improves the efficiency of synchronization by executing at the fastest possible pace permitted by the system characteristics and the active interaction message.

Detection of imminent interactions and the dynamic calculation of the pace of execution are facilitated by the use of a formal model of a distributed simulation system and its dynamics in terms of synchronization. This formalism is based on finite-state automata models of discrete-event simulations. The basic finite state automaton is modified to accommodate timing and communication resulting in what is called a “communicating timed finite-state automaton”.

Using the mathematical representation of a DSS, the requirements of paced execution during an interaction were analyzed and the limiting condition for preservation of synchronization with realistic timing assumptions was derived. The derived limiting condition basically yields an upper-bound on how fast the simulations can advance (i.e. execute future events) during an interaction and is incorporated in the APP algorithm to calculate the pace of executions during interactions. It has been shown that this upper-bound on the speed of execution depends on the following factors:

1. Maximum communication delay in the DSS.
2. Maximum system-clock synchronization error in the DSS.
3. Virtual interaction lag associated with the interaction.

The first two factors are inversely proportional with the upper-bound on the speed of execution while the last factor is directly proportional. This finding is quite intuitive since communication delay and system-clock synchronization error both contribute to the uncertainty in the amount of wallclock time required to complete an interaction, while the virtual interaction lag provides room for execution of future events without violating causality constraints between the interaction event and future events.

Above mentioned findings suggest an important conclusion: the fundamental reason for the occurrence of time-ambiguities in a DSS is non-synchronous execution of distributed simulation processes. This conclusion is based on the fact that existence of a large gap between LVCs of interacting simulation processes increases the risk of occurrence of time-ambiguities. If an entire DSS runs synchronously using paced execution all the time, LVCs of all simulation processes would stay close to each other and the risk of the DSS going out of synchronization would be eliminated.

Paced execution is slower compared to as fast as possible execution, therefore the APP algorithm provides a mechanism to dynamically switch between paced and regular
execution phases depending on the state of the DSS. This mechanism is more efficient than continuous pacing and also continuously guarantees synchronization in the DSS.

The problem of synchronization is likely to keep its place as the most important problem to be addressed in the area of parallel and distributed simulation. However, it is a widely accepted concern among the frontiers of the area that without “plug-and-play” mechanics to develop and deploy distributed simulation systems, the technology would mostly reside in the hands of researchers and will not be widely adopted by businesses.

This research addresses this important problem by creating a formalism along with a mechanism for automated synthesis of distributed discrete-event simulation systems. In this respect, the same mathematical formalism (i.e. communicating timed finite-state automata) mentioned above were adapted to create a representation of simulation models to be used in a distributed setting. This representation captures the structure of a simulation model while ignoring its decision making logic and is named a “structural model”. Structural models form the heart of local synchronization agents that are the main components responsible for synchronization of a DSS in the APP framework.

A local synchronization agent is a computer program that couples with a simulation process by tracking its entities and controlling its execution pace. Local synchronization agents communicate among each other to coordinate synchronization activities according to a set of rules created to preserve synchronization in the DSS.

In this respect, structural models are the only components in the APP synchronization framework that change from implementation to implementation. The rest of the framework is only implemented once for a predefined set of standards governing the protocols of communication and timing in a DSS. As long as these standards are kept fixed, there is no need to re-implement the APP synchronization framework when the simulation processes constituting a DSS change.
This modular design allows isolation of model mapping from the process of synchronization, hence providing a user-friendly environment for distributed simulation developers with limited knowledge of the theory of distributed simulation and associated problems of synchronization.

Model mapping is performed by an independent interpreter (a middle-ware) which is developed to map modeling components of a commercial-off-the-shelf (COTS) simulation solution to the predefined components of structural models: entities, states, events and messages. Usually, development of this middle-ware would require in-depth knowledge of the utilized simulation software and it should be performed by developers who are well-experienced with the simulation software. Ideally, this step would be done by the developer company of the simulation software and could be provided as an add-in to the clients using the software.

Moreover, development of such interpreters for multiple COTS simulation packages would enable users to transparently developed heterogeneous distributed simulation systems where interaction messages are standardized and handled by the APP synchronization framework.

In this research, a limited interpreter was developed for the Arena™ simulation package of Rockwell Software and successfully implemented and executed in a distributed setting to demonstrate the feasibility of the research idea. The development of the interpreter (coding, debugging and verification) took approximately 60 man-hours with no direct help from the developers of the Arena™ simulation package and the developer (i.e. the author) was not a professional software developer. A professional team of software developers with in-depth knowledge of the simulation platform could not only shorten this development cycle substantially but could also create optimized solutions resulting in faster and more seamless execution. This is in general true for the implementation of the APP algorithm and the overall synchronization framework.
The fundamental problem stands as to whether the simulation community is willing to agree on a standard or on multiple standards to represent simulation models and allow interoperability between independent COTS simulation packages. While this is a very important problem that affects the usability of any distributed simulation framework targeting heterogeneous implementation, it is not in the interest of this research to provide an exclusive mechanism for heterogeneous implementation.

Last but not least, the implemented APP synchronization framework was tested using an experimental setup to determine its performance characteristics. Two factors were determined to categorize simulation processes for experimental analysis:

1. State-delay variability of a simulation process.
2. Degree of coupling between distributed simulation processes.

State-delay variability is a measure of the amount of error made in predicting the timing of imminent interactions and was shown to have a converse effect on the performance of the APP synchronization framework. As the variability increases, the synchronization overhead increases resulting in decreasing speed-up compared to the sequential simulation of the same simulation processes.

The degree of coupling between interacting simulation processes is a measure of interdependence between simulation processes. Analyses showed that as the degree of coupling between interacting simulation processes increases, the synchronization overhead increases and thus the efficiency of the APP framework declines.

Both of these factors were shown to affect the performance of the APP synchronization framework. However, it is not possible to calculate universal thresholds for these factors. Therefore, the analyses suggest that the best way to determined speed-up characteristics of the APP synchronization framework is to conduct test runs and compare parallel execution time against sequential execution time for a specific DSS.
It is important to note that, this research does not aim to create a super-efficient synchronization mechanism. Speed-up was not a driving factor in the development of the APP synchronization framework and therefore one must not assess the usability of the APP framework solely based on speed-up characteristics. The APP framework provides a reasonably efficient conservative synchronization mechanism while creating a modular system for rapid development of distributed simulation systems which have many benefits beyond the speed-up due to parallelization.

7.2 Research Contributions

The APP synchronization framework provides an efficient conservative synchronization mechanism and substantially reduces the development cycle of distributed simulation systems by standardizing the repeating components and isolating the non-repeating implementation specific components. In this respect, the APP synchronization framework is the first synchronization framework in its class to provide a modular and user-friendly structure to develop and deploy distributed simulations.

Details of the synchronization mechanism are encapsulated within the local synchronization agents, developers using the APP synchronization framework are not required to know anything about the synchronization mechanism. The developer is only required to create simulation models, convert them into structural models using provided interpreters and determine the operating parameters of the distributed simulation environment such as the communication delay and network bandwidth. The rest is automatically handled by the framework and the development effort associated with parallelization is reduced to a minimum. This is a unique property and is an important contribution to the state of the art of the parallel and distributed simulation research.

The APP framework is based on time-stepped (i.e. paced) coordination of distributed simulations to avoid violation of event causality relations. Although this
technique has been previously hinted in the literature, there are no explicit synchronization mechanisms reported in the parallel and distributed discrete-event simulation literature that is build on pacing based synchronization. In this respect, for the first time in this research such a technique is developed along with necessary formal models and analyses.

Moreover, the APP synchronization algorithm is one of the first such techniques that perform conservative synchronization in an adaptive manner. Most of the existing adaptive synchronization techniques utilize optimistic synchronization protocols as discussed in Chapter 2.

Like any other synchronization mechanism, the APP synchronization framework is also prone to suffer from overhead when used to synchronize a tightly coupled DSS. It would be best to utilize the APP framework for systems with low degree of coupling and also with low processing time variability. In this respect, distributed high-fidelity simulations of automated manufacturing systems and supply chains that are constituted by such systems are among good candidates for efficient use of this framework. Other possible implementation areas are distributed simulation of computer systems and integrated digital circuits conforming to very large-scale integration (VLSI) technology, military battlefield simulations including simulations of very large scale swarm behavior (e.g. swarms of unmanned aerial vehicles).

A critical assumption of the APP framework is the existence of a well-defined maximum communication delay in the communication network used for distributed simulation execution. This assumption mandates the use of dedicated communication networks with strong bandwidth control. The APP framework cannot guarantee conservative synchronization when a general shared network is used for communication such as the Internet, where strong bandwidth control is not possible.
In order to create geographically distributed simulations using the APP framework, dedicated wide area networks are required. Current technology initiatives by developers of networking equipment (e.g. Intel, Cisco etc.) are moving in the direction of creating wide area wireless networks for exclusive use of geographically distributed business enterprises. In fact, cellular communication networks are good examples of such systems.

Although the APP synchronization framework is a conservative mechanism, it is significantly different from the Chandy-Misra-Bryant synchronization algorithm. As discussed in Chapter 2, Chandy-Misra-Bryant synchronization mechanism is the root of almost all basic conservative synchronization protocols and also implemented as part of the time-management service of the High Level Architecture (HLA).

The Chandy-Misra-Bryant algorithm depends on coordinated sequential execution of events in the DSS during interactions, while the APP algorithm uses paced concurrent execution during interactions resulting in deadlock-free execution. The Chandy-Misra-Bryant algorithm was shown to be prone to deadlocks and several techniques have been developed to detect and overcome deadlocking. In the APP framework, since execution does not stop during interactions, deadlocking is not possible. Therefore, deadlock detection is not required resulting in elimination of some of the synchronization overhead present the Chandy-Misra-Bryant algorithm.

However, paced execution is slower compared to as fast as possible execution and depending on the number of events being executed during an interaction and the state-delay variability of the simulation processes, the relative performance of the APP algorithm can be arbitrarily better or worse than the Chandy-Misra-Bryant algorithm and any of its derivatives.

The APP algorithm is also significantly different from the Time-Warp synchronization algorithm. First and foremost, Time-Warp is an optimistic protocol and
the APP algorithm is a conservative one. In this respect, philosophies of both techniques are completely different as well as their strengths and weaknesses. Optimistic techniques are well known for their superior performance in very loosely coupled systems. Especially, if interactions are very rare events, optimistic techniques such as Time-Warp and its derivatives are known to perform more efficiently compared to conservative techniques.

However, it is well known that [Srinivasan and Reynolds, 1995] neither conservative nor optimistic synchronization methods are universally more efficient than the other. The same conclusions have been reached for adaptive versions of the Time-Warp. Research has shown that synchronization techniques can arbitrarily outperform each other most probably depending on the degree of coupling and other factors which cannot be directly and universally measured.

By the same token, it is intuitively possible to set up experiments where the APP framework and the Time-Warp technique can arbitrarily outperform each other. An immediate example is a DSS in which there are several not very loosely-coupled simulation processes so that even a single causality error would require a massive roll-back to be propagated in the DSS requiring a collective processing time and communication overhead far more than that is required by the pacing of the execution during a single interaction. On the other hand, a DSS execution with no actual interactions but only potential ones would give advantage to the Time-Warp mechanism.

Nevertheless, one advantage of the APP framework over all of the optimistic synchronization frameworks is the ease of development. Optimistic protocols require mechanisms to roll-back simulation executions in the event of a causality violation.

A roll-back is basically implemented by rewinding the execution steps one event at a time from an event stack. Thus, during an execution, every event that is executed must be kept in a stack until it is known for sure that a roll-back is not possible to reach
that event (see Chapter 2 for more detail). Moreover, if a rolled-back event has affected another simulation process, that simulation process must also perform a roll-back, resulting in possible propagation of roll-backs throughout an entire DSS. All of these operations require the synchronization system to have full control of the simulation kernel (the main program that runs the simulation) including the ability to reverse the execution which is not a standard capability included in any mainstream COTS simulation software.

For this reason, all of the optimistic synchronization techniques have been implemented with their own specialized simulation kernels. There is no optimistic distributed simulation system reported in the literature without a dedicated simulation kernel. This is a clear indication of the high degree of experience and knowledge required to implement such a system. For these same reasons, it is not possible to find a publicly available and stable implementation of the Time-Warp synchronization technique that is compatible with any mainstream commercial simulation package, while the idea of Time-Warp has been around for almost two decades.

On the other hand, the APP framework only requires three basic capabilities regarding the simulation kernel:

1. Ability to read the state of an entity (current location in the simulation process).
2. Ability to pause/resume simulation execution at a certain virtual time.
3. Ability to read the event calendar.

Notice that none of these capabilities requires a deviation from the standard event-driven execution of a discrete-event simulation process. Many simulation solutions such as the Arena™ simulation package of Rockwell Automation and the AutoMod™ simulation package of Brooks Automation already provide this sort of functionality and more to the user through a well-defined programming interface similar to those found in standard office applications such as Microsoft Word™ and Excel™.
It is argued here that the Adaptive Partial Pacing based synchronization framework introduced in this dissertation is a viable alternative to other synchronization frameworks and can be further developed to be widely adopted by academic and commercial use. Especially, the ability of the APP framework to be automatically integrated with a COTS simulation package provides the possibility of customization of the APP framework to integrate other existing synchronization techniques with COTS simulation packages.

7.3 Areas of Improvement and Directions for Future Research

One of the main problems faced during the development of the APP framework is to provide a highly accurate timing source. Accurate timing is crucial since coordination and speed of pacing depends very much on distributed timing in the DSS. The resolution of the clocks used for timing determines the maximum speed of pacing.

Although modern computer processors provide very high resolution hardware driven real-time clocks, it is not possible to read these hardware clocks at their best resolution due to the timing uncertainties involved in multi-tasking operating systems.

In a multi-tasking operating system such as Microsoft Windows or Linux, multiple processes are executed sequentially for very short time-slices creating the illusion of concurrent execution of multiple processes. In reality, the number of processes running concurrently in a computer cannot exceed the number of processors available.

Almost all of the multi-tasking operating systems use an arbitrary scheduling scheme to order execution of multiple processes. Since the order of execution is arbitrary, there is no guarantee whether a process is executed at regular time intervals. Therefore, any process that reads the hardware clock of the computer is prone to a nondeterministic amount of execution delay, which means that although the hardware clock is high
resolution the process reading the clock is not executed as frequently and regularly as the
clock resolution.

This fact suggests that the operating system imposes another level of clock
resolution which is far less than that of the computer’s hardware clock. In the case of
Microsoft Windows XP operating system, the resulting best available clock resolution is
around 1 milliseconds, which results in the fastest pacing of 1000 virtual time steps per
one real time step assuming that the smallest unit of virtual time used in a simulation is
seconds.

On the other hand, the problem of clock resolution is not unsolvable. Another
class of operating systems called “real-time” operating systems is available and they are
specifically designed for time-critical systems such as high-speed machine tool control.
In real-time operating systems, processes are scheduled deterministically with well-
known and adjustable time-slices. An example of such an operating system is RTOS
which is based on the Linux operating system. There is also a real-time subsystem named
RTX developed by Venturcom which runs with the Windows XP platform.

Since clock resolution has a profound effect on the speed of pacing, it would be
useful to test the APP framework using a real-time operating system in order to assess
full performance of the framework. Considering that hardware clock resolutions of 100
nanoseconds are easily available with current processors, a speed gain of $10^4$ could be
attainable using a real-time operating system.

On the more extreme side, the local synchronization agents can be implemented
on a dedicated piece of hardware with self timing and clock-synchronization capabilities
and communicate with the coupled simulation processes via an available high speed
communication bus in a way similar to high-end computer graphics cards which handle
graphics operations using its own dedicated graphics processor and decrease the burden
on the computer processor.
Another important area of improvement is development of full-scale mappings for multiple COTS simulation packages and creating a standard for heterogeneous implementation of the APP framework. Such standardization would open a gateway for interoperability between COTS simulation packages which is currently a very popular and important research issue.
References


Ferscha, A., 1995, Probabilistic adaptive direct optimism control in time warp. Proceedings of the 9th Workshop on Parallel and Distributed Simulation (PADS'95), Lake Placid, NY, USA, p. 120-129.


Prabhu, V. V. and Duffie, N. A., 1995, Distributed simulation approach for enabling cooperation between entities in heterarchical manufacturing


Appendix A
Overview of Arena™ Modeling Constructs

An Arena™ simulation project is composed of two main parts:

1. Definitions
2. Model

Definitions are specified using special constructs called elements. Each element defines a part of the simulation process. Commonly used elements within the context of this study and their brief descriptions are provided in Table A-1.

Elements are used to declare the common components of a simulation model such as entity types, resource names and characteristics, queues and queue disciplines, and so on.

An Arena™ model is built using the basic modeling constructs called modules. Modules are represented by polygonal shapes in the Arena™ modeling environment. Modules are categorized into panels according to their functionalities. For example, PROCESS module represents the processing of entities in the model and it is a member of the Basic Processes panel. TRANSPORT is another module that represents transfer of entities from one station to another in the model but it is a member of the Advanced Transfer panel.
Table A-1: Important element modules of Arena™ simulation package

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENTITIES</td>
<td>The ENTITIES element defines entity types (e.g., part types, order types, customer types) that may be assigned to entities in the model.</td>
</tr>
<tr>
<td>ATTRIBUTES</td>
<td>The ATTRIBUTES element specifies the total number of general-purpose entity attributes, their names, and attribute initial values.</td>
</tr>
<tr>
<td>RESOURCES</td>
<td>The RESOURCES element defines the characteristics of the resources, including the resource names and initial capacities.</td>
</tr>
<tr>
<td>VARIABLES</td>
<td>The VARIABLES element specifies the total number of global variables, their names and, if desired, initial values.</td>
</tr>
<tr>
<td>MESSAGES</td>
<td>The MESSAGES element defines message strings that simulation entities may send to an external process when Arena is running in execution mode.</td>
</tr>
<tr>
<td>QUEUES</td>
<td>The QUEUES element specifies the total number of queues along with their names, ranking criterion, and associated model QUEUE block. Ranking Criterion specifies the order for a single queue.</td>
</tr>
<tr>
<td>ARRIVALS</td>
<td>The ARRIVALS element creates batches of entities that arrive at the system model at specified times. If Message is specified, then a single batch of entities is created if Arena is running in execution mode and a message number MessageType is received, where MessageType is a positive integer.</td>
</tr>
<tr>
<td>REPLICATE</td>
<td>The REPLICATE element specifies the number of simulation replications, the beginning time of the first replication, the maximum length or terminating condition for each replication, the type of initialization to be performed between replications, and the time period after the beginning of the run at which statistics are to be cleared.</td>
</tr>
<tr>
<td>PROJECT</td>
<td>The PROJECT element is used to label the Summary Report. This report is a statistical summary of the simulation responses for each replication. It is automatically generated at the end of each simulation replication.</td>
</tr>
</tbody>
</table>

As mentioned earlier, we selected a subset of the Arena™ modules for our selected implementation domain – modeling of queuing networks. Selected modules and their brief explanations (borrowed from Arena™ 7.0 help files) are provided in Table A-2.

Modules have predefined input and output connection points. They can be dragged and dropped on a model window and one can graphically connect modules together to define the flow of entities in the model.

Data required for modules can be entered in special dialog boxes which can be displayed by double clicking on module graphics.
Entities visit modules sequentially following the paths defined by the interconnections among the modules. An entity starts its flow at the CREATE or ENTER module. CREATE modules generate new entities, while ENTER modules accept incoming entities that are being transferred from other stations.

In a single model (non-distributed) setting all stations are local. Use of stations locally is a way of modeling distinct territories of a system. However, in a distributed setting one can define stations which are located in remote simulation models. This is done by sending and receiving messages defined in the MESSAGES element.

The ARRIVALs element can be configured to respond to unsolicited messages received from processes external to the simulation process (these processes can be residing in another computer connected to the common network).

Arena™ has a built-in server-client type socket communication mechanism for sending and receiving predefined message strings over a local or wide area network. An add-on called “Arena™ Real-Time” (a.k.a. Arena™-RT) is required for this functionality. Arena™-RT also allows users to control the speed of execution with respect to real-time by specifying a speed factor.

Real-time functionality of Arena™ is especially useful for industrial control purposes where real-time execution is required. Obviously, it is a desired property for the purposes of this study. More information on this issue is provided in Chapter 6 along with the implementation of the adaptive synchronization algorithm.
### Table A-2: Important modules of the Arena™ simulation package

<table>
<thead>
<tr>
<th>Class</th>
<th>Module</th>
<th>Description</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial Module</td>
<td>CREATE</td>
<td>This module is intended as the starting point for entities in a simulation model. Entities are created based on a time between arrivals. Entities then leave the module to begin processing through the system. The entity type is specified in this module.</td>
<td><img src="create.png" alt="Create" /></td>
</tr>
<tr>
<td></td>
<td>Basic Process Panel</td>
<td>The ENTER module defines a station corresponding to a location where an entity arrives at in the local logical process. An unsolicited message triggers a pre-specified entity type to start from an ENTER module. The message structure, entity type and the name of the ENTER module are defined in the ARRIVALS element.</td>
<td><img src="enter.png" alt="Enter" /></td>
</tr>
<tr>
<td>Internal Module</td>
<td>PROCESS</td>
<td>This module is intended as the main processing method in the simulation. Options for seizing and releasing resource constraints are available. The process time is allocated to the entity.</td>
<td><img src="process.png" alt="Process" /></td>
</tr>
<tr>
<td></td>
<td>Basic Process Panel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decision / Auxiliary Module</td>
<td>DECIDE</td>
<td>This module allows for decision-making processes in the system. It includes options to make decisions based on one or more conditions or based on one or more probabilities. Conditions can be based on attribute values, variable values, the entity type, or an expression.</td>
<td><img src="decide.png" alt="Decide" /></td>
</tr>
<tr>
<td></td>
<td>Basic Process Panel</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ASSIGN</td>
<td>This module is used for assigning new values to variables, entity attributes, entity types, entity pictures, or other system variables. Multiple assignments can be made with a single Assign module.</td>
<td><img src="assign.png" alt="Assign" /></td>
</tr>
<tr>
<td></td>
<td>Basic Process Panel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Final Module</td>
<td>TRANSPORT</td>
<td>The TRANSPORT module transfers the controlling entity from one station to another. If Arena is running in execution mode (sends and receives messages), the module will send a pre-specified message defined in the MESSAGES element upon entering the TRANSPORT module.</td>
<td><img src="transport.png" alt="Transport" /></td>
</tr>
<tr>
<td></td>
<td>Advanced Transfer Panel</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DISPOSE</td>
<td>This module is intended as the ending point for entities in a simulation model. Entity statistics may be recorded before the entity is disposed.</td>
<td><img src="dispose.png" alt="Dispose" /></td>
</tr>
</tbody>
</table>
After entities start their flow in the simulation, they go through steps of processing modeled by the PROCESS modules. There are four types of PROCESS modules:

1. Delay
2. Seize-Delay
3. Seize-Delay-Release
4. Delay-Release

The keyword **seize** corresponds to the event of reserving a resource for processing. **Release** corresponds to the freeing of the resource once the processing is done and resource is available for other entities to reserve (seize). Seize and release keywords control the temporary ownership of a resource by entities. The keyword **delay** corresponds to the activity of actual processing of entities. Notice that while seize and release correspond to events, delay models an activity.

When a type with seize and/or release keyword is used, that means one or more resources must be associated with the PROCESS module. Therefore, type 1 PROCESS modules model activities that do not require use of a resource. A PROCESS module of type 2, 3 or 4 has an internal input queue which holds incoming entities in case the associated resource is not available. On the other hand, a PROCESS module in general does not have an internal output queue. Once an entity is done processing, it will immediately leave the module and arrive at the next connected module.

In this implementation, we only use **seize-delay-release** type PROCESS modules. Also, we assume that each PROCESS module models a single server with an input queue, meaning that there is one unique resource associated with each PROCESS module. The internal input queue has unlimited capacity and the queuing discipline is FIFO (first-in-first-out). The resource capacity is one and preemption is not allowed.
The processing time can be configured as a random variable sampled from one of several available probability distributions (Normal, Uniform, Beta, Triangular etc.) or it can be a constant. One can parameterize the processing time so that different entity types have different processing times on the same resource. This is done by using entity type attribute as a parameter in the processing time expression.

In this implementation, we restricted the processing time variables to be sampled only from a probability distribution with finite and non-zero lower and upper bounds. Uniform and Triangular distributions with non-zero bounds are mostly used. Also, a general probability distribution can be used by truncating its extreme ends. Obviously, constant processing times are also permitted on the condition that they are strictly positive.

This restriction on processing times is due to a limitation in the synchronization framework, in particular stemmed from the mechanism that predicts future external events. This issue is further explained in the next chapter (Chapter 5) where we describe the synchronization algorithm.

In general, any time delay in the simulation processes must obey the above defined rules. Therefore, interarrival times configured in CREATE modules and transportation times configured in TRANSPORT modules are also sampled from either the Uniform or the Triangular probability distributions with non-zero bounds.

The base time unit is defined in the REPLICATIONS element, which can be one of hours, minutes or seconds. We selected seconds as the base time unit for the models.

Once the flow of an entity is over, it is terminated at either a DISPOSE module or a TRANSPORT module. DISPOSE modules simply discard the entity. This method of termination is used for entities which do not travel to other logical processes. Entities that
are destined for another logical process terminate at a TRANSPORT module, which sends a message to the target logical process.

Apart from the modules explained above, there are two modules which are included in the simulation models but not mapped to structural model components. These are DECIDE and ASSIGN modules.

When an entity arrives at a DECIDE module, the condition(s) at the module is evaluated and depending on which condition(s) are true, the entity is directed towards one of the connected modules. DECIDE modules represent a major part of the decision making in the simulation model. Conditions of a DECIDE module can be anything related to the state of the visiting entity or the state of the system resources in general or just a random evaluation.

The ASSIGN module is used for assigning values to entity attributes and global variables. It is an auxiliary module and does not have any affect on the flow of entities.
Appendix B

Mapping of Arena™ Elements to Structural Model Components

For each of the selected modules listed in Appendix A, a mapping to the components of a structural model is defined. Depending of the module type and its connections to other modules, mapped components can be assembled to form the structural model of the simulation process.

Notice that the Arena™ modules are characterized into four classes in Table A-2 in Appendix A:

1. Initial modules – these are the modules where entities start their flow in the model.
2. Final modules – arrival of an entity at a final module means the entity has finished its flow in the local model and is due to be terminated or transferred to another model (station).
3. Internal modules – these are the modules that entities visit between initial and final modules. Also, these modules have a corresponding state/event representation at the structural model.
4. Decision/Auxiliary modules – these are also internal modules by definition however they are not represented in the structural model since they corresponds to peripheral operations (such as variable assignments) or decision operations.

This classification is directly related to the classification of states in the structural model. There are initial states where entities start from, there are final states where entities end their flow and in between there are internal states. Therefore, there is a one-to-one relationship between state classes and module classes.
We can also classify modules by message sending/receiving capabilities. Notice that modules TRANSPORT and ENTER has direct or indirect relations with incoming and outgoing messages. These messages correspond to entities being transferred between processes. Similarly, events in the structural model are classified as input, output and internal events, again showing a direct relationship with the modules’ messaging capabilities.

Based on these two relationships between Arena™ modules and structural model components we defined the mappings depicted in Table B-1. A legend for the graphical representation of structural model components is provided in Figure B-1.

Initial module class members CREATE and ENTER are mapped to two states defined as follows:

- **State \( \chi_i \)** - this is the initial state where a potential virtual entity waits for a scheduled creation or entrance to the model.
- **State \( \chi_{i+1} \)** - this is the internal state at which an entity arrives immediately after it is created or it entered the model. This state is not connected at its output and can be connected to any event.

These two states are connected to each other by an event \( \sigma^{(i,i+1)}_{\text{create}} \) or \( \sigma^{(i,i+1)}_{\text{enter}} \) which represents the event of creation or entrance to the model. In the case of an ENTER module, there is also an input message \( m^{(i,i+1)}_{\text{enter}} \) attached to the event \( \sigma^{(i,i+1)}_{\text{enter}} \). This is an unsolicited input message received from a remote logical process which triggers (or schedules) the input event \( \sigma^{(i,i+1)}_{\text{enter}} \). On the other hand, the event \( \sigma^{(i,i+1)}_{\text{create}} \) is an internal event since it has no attached messages.

Internal module class member PROCESS is mapped to an event \( \sigma^{(i,j)}_{\text{process}} \) which is connected to a state \( \chi_i \). State \( \chi_i \) is an internal state that represents processing of the visiting entity including the waiting in the internal queue prior to actual processing.
Table B-1: Mapping of Arena™ modules to Structural Model Components

<table>
<thead>
<tr>
<th>Module Class</th>
<th>Arena Module</th>
<th>Structural Model Equivalent</th>
<th>Component Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial Module</td>
<td>CREATE</td>
<td>![Diagram of CREATE module]</td>
<td>( \chi_i \in X_{\text{Initial}} ) ( \sigma^\text{create}<em>{(i,i+1)} \in \Sigma</em>{\text{Initial}} )</td>
</tr>
<tr>
<td></td>
<td>ENTER</td>
<td>![Diagram of ENTER module]</td>
<td>( \chi_i \in X_{\text{Initial}} ) ( \sigma^\text{enter}<em>{(i,i+1)} \in \Sigma</em>{\text{Initial}} )</td>
</tr>
<tr>
<td>Internal Module</td>
<td>PROCESS</td>
<td>![Diagram of PROCESS module]</td>
<td>( \chi_i \in X_{\text{Internal}} ) ( \sigma^\text{process}<em>{(i,i)} \in \Sigma</em>{\text{Internal}} )</td>
</tr>
<tr>
<td>Decision / Auxiliary Module</td>
<td>DECIDE</td>
<td>![Diagram of DECIDE module]</td>
<td>Not modeled</td>
</tr>
<tr>
<td></td>
<td>ASSIGN</td>
<td>![Diagram of ASSIGN module]</td>
<td>Not modeled</td>
</tr>
<tr>
<td>Final Module</td>
<td>TRANSPORT</td>
<td>![Diagram of TRANSPORT module]</td>
<td>( \chi_i \in X_{\text{Final}} ) ( \sigma^\text{transport}<em>{(i,i)} \in \Sigma</em>{\text{Output}} )</td>
</tr>
<tr>
<td></td>
<td>DISPOSE</td>
<td>![Diagram of DISPOSE module]</td>
<td>( \chi_i \in X_{\text{Final}} ) ( \sigma^\text{dispose}<em>{(i,i)} \in \Sigma</em>{\text{Internal}} )</td>
</tr>
</tbody>
</table>
Figure B-1: Graphical Representations of Structural Model Components
Final module class members TRANSPORT and DISPOSE are mapped similar to the PROCESS module. An event \( \sigma_{\text{transport}}^{(s,i)} \) or \( \sigma_{\text{dispose}}^{(s,i)} \) and a final state \( \chi_i \) where the entity gets terminated. For the TRANSPORT module, the event \( \sigma_{\text{transport}}^{(s,j)} \) is an output event with the attached message \( m_{\text{transport}}^{(s,j)} \) destined for a remote logical process. The transfer of the entity is simulated by terminating the entity at the local process and recreating it at the target logical process by sending the unsolicited message \( m_{\text{transport}}^{(s,j)} \).

The trailing events in the mapping of the PROCESS, TRANSPORT and DISPOSE modules are not connected to a source state. During the synthesis of a structural model these events are connected to other mapped states based on the sequence defined in the simulation model. In a sense, these basic elements are assembled together to create structural models.

DECIDE and ASSIGN modules are not mapped to any structural model components. They are simply replaced by direct connections between source and sink modules connected to them.
Appendix C

Data Structures for Structural Models

C.1 StructuralModel Package Member Classes

EntityStructuralModel Class
EntityType Class
Event Class
EventType Class
EventSet Class
InteractionMessage Class
Message Class
State Class
StateTimeTableItem Class
StateType Class
StateSet Class
TransitionMap Class
C.2 EntityStructuralModel Class

Represents entity-specific structural models.

**Access:** Public  
**Base Classes:** Object

<table>
<thead>
<tr>
<th>Members</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ModelStates</td>
<td>State set of the entity-specific structural model object.</td>
</tr>
<tr>
<td>ModelEvents</td>
<td>Event set of the entity-specific structural model object.</td>
</tr>
<tr>
<td>ModelEntity</td>
<td>Entity type of the entity-specific structural model object.</td>
</tr>
<tr>
<td>ModelTransitionMap</td>
<td>Transition map of the entity-specific structural model object.</td>
</tr>
<tr>
<td>ModelID</td>
<td>A unique numerical identifier.</td>
</tr>
</tbody>
</table>

C.3 EntityType Class

Represents entity types in a simulation process/model.

**Access:** Public  
**Base Classes:** Object

<table>
<thead>
<tr>
<th>Members</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EntityTypeID</td>
<td>A unique numerical identifier.</td>
</tr>
<tr>
<td>EntityName</td>
<td>Name of the entity type as defined in the simulation model.</td>
</tr>
</tbody>
</table>
C.4 Event Class

Represents events in a structural model.

**Access:** Public

**Base Classes:** Object

<table>
<thead>
<tr>
<th>Members</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EventID</td>
<td>A unique numerical identifier.</td>
</tr>
<tr>
<td>Type</td>
<td>Type of the event object as defined in the EventType enumeration type.</td>
</tr>
<tr>
<td>EventMessage</td>
<td>Represents the message object (if any) associated with the event.</td>
</tr>
<tr>
<td>SinkState</td>
<td>Represents the state object that the event terminates at.</td>
</tr>
<tr>
<td>SourceState</td>
<td>Represents the state object that the event originated from.</td>
</tr>
<tr>
<td>EnabledEntityTypes</td>
<td>Enabled entity types.</td>
</tr>
</tbody>
</table>

C.5 EventType Enumeration

Types of Event objects.

**Access:** Public

<table>
<thead>
<tr>
<th>Members</th>
</tr>
</thead>
<tbody>
<tr>
<td>InputEvent</td>
</tr>
<tr>
<td>InternalEvent</td>
</tr>
<tr>
<td>OutputEvent</td>
</tr>
</tbody>
</table>
C.6 EventSet Class

Represents the event set of a structural model.

**Access:** Public

<table>
<thead>
<tr>
<th>Members</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EventSetID</td>
<td>A unique numerical identifier.</td>
</tr>
<tr>
<td>EventList</td>
<td>List of events.</td>
</tr>
<tr>
<td>NumberEvents</td>
<td>Total number of events.</td>
</tr>
<tr>
<td>OutputEventList</td>
<td>Sub-list of output events.</td>
</tr>
</tbody>
</table>

C.7 InteractionMessage Class

Represents true interaction messages exchanged among logical processes.

**Access:** Public

**Base Classes:** Message

<table>
<thead>
<tr>
<th>Members</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VirtualInteractionLag</td>
<td>Virtual interaction lag of the message object.</td>
</tr>
<tr>
<td>MessageID</td>
<td>Inherited from Message class.</td>
</tr>
<tr>
<td>OriginLPID</td>
<td>Inherited from Message class.</td>
</tr>
<tr>
<td>TargetLPID</td>
<td>Inherited from Message class.</td>
</tr>
<tr>
<td>TimeStamp</td>
<td>Inherited from Message class.</td>
</tr>
<tr>
<td>EntityTypeID</td>
<td>Entity type ID of the interaction message.</td>
</tr>
<tr>
<td>EntityID</td>
<td>ID of the owning entity, used only when the interaction message is transmitted to another LP.</td>
</tr>
</tbody>
</table>
C.8 Message Class

Represents a generic message.

**Access:** Public

**Base Classes:** Object

<table>
<thead>
<tr>
<th>Members</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_messageID</td>
<td>A unique numerical identifier.</td>
</tr>
<tr>
<td>_timestamp</td>
<td>Represents the timestamp and must be redefined by the subclasses for each specific context.</td>
</tr>
<tr>
<td>_originLPID</td>
<td>Unique identifier of the originating logical process.</td>
</tr>
<tr>
<td>_targetLPID</td>
<td>Unique identifier of the destination logical process.</td>
</tr>
</tbody>
</table>

C.9 State Class

Represents the state of a structural model.

**Access:** Public

**Base Classes:** Object

<table>
<thead>
<tr>
<th>Members</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StateID</td>
<td>A unique numerical identifier.</td>
</tr>
<tr>
<td>Type</td>
<td>Type of the state as defined in the StateType enumeration type.</td>
</tr>
<tr>
<td>LowerTimeBound</td>
<td>Lower time bound.</td>
</tr>
<tr>
<td>UpperTimeBound</td>
<td>Upper time bound.</td>
</tr>
<tr>
<td>EnabledEntityTypes</td>
<td>Enabled entity types.</td>
</tr>
<tr>
<td>StateTimeTable</td>
<td>Time table that holds minimum virtual time to firing of each accessible interaction message.</td>
</tr>
</tbody>
</table>
C.10 StateTimeTable Class

Represents items of the StateTimeTable.

**Access:** Public

**Base Classes:** Object

<table>
<thead>
<tr>
<th>Members</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MinVirtualTime</td>
<td>Minimum virtual time required to pass before the interaction message fires.</td>
</tr>
<tr>
<td>ValidEntityType</td>
<td>The entity type object for which the state time table item is valid.</td>
</tr>
<tr>
<td>Message</td>
<td>Reference to the interaction message object.</td>
</tr>
</tbody>
</table>

C.11 StateType Enumeration

Types of state objects.

**Access:** Public

<table>
<thead>
<tr>
<th>Members</th>
</tr>
</thead>
<tbody>
<tr>
<td>InitialState</td>
</tr>
<tr>
<td>InternalState</td>
</tr>
<tr>
<td>OutputState</td>
</tr>
<tr>
<td>FinalState</td>
</tr>
</tbody>
</table>
C.12 StateSet Class

Represents the set of states of a structural model.

**Access:** Public

**Base Classes:** Object

<table>
<thead>
<tr>
<th>Members</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StateSetID</td>
<td>A unique numerical identifier.</td>
</tr>
<tr>
<td>StateList</td>
<td>List of states.</td>
</tr>
<tr>
<td>NumberStates</td>
<td>Total number of states.</td>
</tr>
<tr>
<td>InitialStateList</td>
<td>Sub-list containing only the initial states.</td>
</tr>
</tbody>
</table>

C.13 TransitionMap Class

Represents the transition function of a structural model.

**Access:** Public

**Base Classes:** Object

<table>
<thead>
<tr>
<th>Members</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TransitionMapID</td>
<td>A unique numerical identifier.</td>
</tr>
<tr>
<td>TransitionMatrix</td>
<td>Transition map matrix.</td>
</tr>
</tbody>
</table>
Appendix D

Glossary

A

Active interaction message (AIM): A potential output message that represents an interaction message in transmission or about to be transmitted.

Adaptive partial pacing (APP): The synchronization method developed in this research, which conservatively protects synchronization by adaptively pacing simulation processes’ execution only during interactions.

B

Bound-map: A collection of a lower and an upper bound on the amount of delay an entity will occur at a specific state.

C

Causality violation: Violation of a casual relationship (before/after relationship) between events of a simulation process.

Communication module (CM): The module of a local synchronization agent that is responsible for communication with remote local synchronization agents over a communication network.

Communicating timed state automaton: A finite state automaton modified by the addition of mathematical structures for timing of events and for interaction with other state automaton.

Conservative synchronization: Generic name for all synchronization methods that guarantee continuous synchronization by controlling the execution of unsafe events (events that may cause a time ambiguity) during run-time.
**D**

**Degree-of-coupling**: A measure of interdependence between entities of a distributed system.

**Distributed simulation system (DSS)**: A collection of at least two interacting simulation processes that run on different computers that are connected via a communication network along with necessary services to manage the simulation processes.

**E**

**Entity-specific structural model**: A version of a general structural model that is specifically reduced to represent the flow logic for a specific entity type of a simulation process.

**Event**: A simulation representation of an action that occurs in a physical process.

**Execution**: An orderly processing of events in a simulation process.

**Execution trace**: An ordered list of events occurred during an execution of a simulation process.

**External event**: An input or an output event.

**F**

**Finite state automaton**: A mathematical model composed of discrete states, events and a transition map which is used to model the non-timed behavior of discrete-event systems.

**Finite state machine**: see Finite state automaton.

**Function**: A well-defined part of the Adaptive partial pacing algorithm which is called from within another part of the APP algorithm.

**G**

**General structural model**: A communicating timed state automaton representation of the flow logic of a simulation process.
**Global positioning system (GPS):** A system using satellites, receivers and software to allow users to determine their exact geographic position as well as the universal time coordinated.

**Input Event:** A remote event scheduled by the receipt of an interaction message from a remote simulation process.

**Interaction management engine (IME):** The engine in a local synchronization agent that manages the prediction of potential output messages, tracks potential interaction messages and controls timely dissemination of potential interaction messages.

**Interaction message:** see Message.

**Internal Event:** An event that is not an external event.

**Internet protocol (IP):** The communications protocol underlying the Internet.

**Leader:** The local synchronization agent linked to the simulation process which is the receiver for a specific interaction message.

**Local event:** A locally created and scheduled event at a simulation process.

**Local virtual clock (LVC):** Clock of a simulation process used to track timing relationships between events during execution.

**Local virtual time (LVT):** The value of a local virtual clock at an instant during execution.

**Logical entity:** see Simulation entity.

**Logical interaction delay (LID):** A representation of physical interaction delay in virtual time units in a simulation process.

**Logical process (LP):** see Simulation process.

**Loosely-coupled distributed system:** A distributed system in which distributed entities constituting the system are independent from each other and occasionally
interact at a substantially lower rate compared to tightly-coupled distributed systems.

**M**

**Message:** A compact piece of information that represents the specifics of a particular interaction between two logical processes.

**N**

**Network bandwidth:** The maximum amount of data that can travel a communications path in a given time, usually measured in bits per second.

**Network time protocol (NTP):** A protocol that provides a reliable way of transmitting and receiving the time over the TCP/IP networks.

**O**

**Optimistic synchronization:** Generic name for all synchronization methods that do not guarantee continuous synchronization during run-time but correct execution by rolling-back simulation processes upon occurrence of time-ambiguities.

**Output event:** An event that fires an interaction message which in turn schedules a matching input event at a remote simulation process.

**P**

**Physical entity:** Entities exchanged between physical processes to interact.

**Physical interaction delay (PID):** The amount of real-time required for an interaction between physical processes to take place.

**Physical process (PP):** A real or imaginary system that is represented by a simulation model and emulated by a simulation process.

**Potential interaction message (PIM):** A data structure that represents a potential interaction in the distributed simulation system.
**Potential output message (POM):** A data structure that represents a potential output message (or the matching output event) and is used to track timing of locally originating potential interactions.

**R**

**Remote event:** An event scheduled by a remote simulation process but created locally.

**S**

**Service:** A well-defined part of the Adaptive partial pacing algorithm which runs continuously during execution to monitor certain conditions and act on them as required.

**Sequential simulation:** Simulation that takes place on a single computer (processor).

**Simulation control engine (SCE):** The engine in a local synchronization agent that is responsible for reading entity states from the linked simulation process and controlling simulation execution pace of the linked simulation process during interactions.

**Simulation entity:** A representation of a physical entity in a simulation process.

**Simulation model:** An abstraction of a real or an imaginary system, which is developed usually with the intention of analyzing the system’s behavior.

**Simulation language:** The formalism used to create simulation models.

**Simulation process:** A computer program that implements a simulation model.

**Speed-up:** The percentage gain in processing time by parallelizing a sequential simulation process.

**State:** A variable that represents the position of a simulation entity in a simulation process.

**State-delay:** The amount of virtual time an entity is sustained at a specific state.
State-table: A timing table attached to a specific state that lists minimum amount of virtual time required for an entity to reach all accessible output events of the local simulation process.

Straggler event: A remote event that causes a time-ambiguity.

Structural Model: see General structural model.

Structural model core (SMC): The part of the interaction management engine of a local synchronization agent that holds all of the entity-specific structural models corresponding to the linked simulation process.

Synchronization: The act of coordinating distributed simulation processes (of a distributed simulation system) to protect the causal ordering of all events in an execution.

Synchronization message: A compact piece of information exchanged between synchronization agents via a communication network to coordinate Adaptive partial pacing synchronization activities.

Tightly-coupled distributed system: A distributed system in which there exists a high degree of interdependence/connectivity between the distributed entities.

Time ambiguity: Violation of a causal relationship between events of a simulation process due to lack of synchronization.

Time management: see Synchronization.

Timestamp: Scheduled local virtual time of execution for an event.

Timing and clock synchronization module (TCSM): The module of a local synchronization agent that is responsible for timing services including the synchronization of system clock of the host computer with a reference wallclock.

Transmission control protocol (TCP): The protocol used in conjunction with Internet protocol to transmit information over the Internet in the form of units.
Universal Time Coordinated (UTC): A time-scale which forms the basis of a coordinated dissemination of standard frequencies and time signals throughout the world (a.k.a. Greenwich Mean Time or GMT).

Virtual interaction lag (VIL): see Logical interaction delay.

Virtual time: The representation of time in a simulation process.

Wallclock: A real-time clock.
Bertan Altuntas received his B.S. degree in Industrial Engineering from The Middle East Technical University, Turkey in 1999. He spent one year in the same department as a teaching assistant until he was awarded the Fulbright fellowship to continue his graduate studies in the U.S. In 2002, he received his M.S. degree in Industrial Engineering and Operations Research from The Pennsylvania State University. He was awarded the Material Handling Education Foundation scholarship in 2002. In 2004, he received his Ph.D. in Industrial Engineering from The Pennsylvania State University. His research interests include parallel and distributed discrete event simulation, evolutionary distributed systems, cellular automata, intelligent computer integrated manufacturing and control of discrete event systems. He is a member of IEEE, INFORMS, IIE and SCS.