NANYANG TECHNOLOGICAL UNIVERSITY

Synchronization Issues in Distributed Simulation Using Commercial Off-The-Shelf (COTS) Simulation Packages

Ph.D Thesis

by

Wang Xiaoguang

Supervisor: Dr. Stephen John Turner

Division of Computer Science
School of Computer Engineering
Nanyang Technological University
Singapore 639798

January 2006

ABSTRACT

During the last few decades, a variety of Commercial Off-The-Shelf (COTS) simulation packages (referred to in this thesis as CSPs) have been developed to facilitate the creation of simulation models. However, most of them are specialized for certain purposes and market niches, and cannot be directly used to build heterogeneous distributed simulations. The advent of the High Level Architecture (HLA) standard makes it possible to connect distributed model components together. The model components can be developed using specific CSPs best suited to the application area. Although there are some successful examples of distributed simulations with CSPs, a general solution to this problem of heterogeneous integration is missing.

The objective of this research is to develop a generic interface between the CSP and the HLA Runtime Infrastructure which can then be tailored to specific CSPs. By integrating with the HLA Runtime Infrastructure through the interface, the CSPs allow the modelers to design their model components in a plug & play manner (involving only process modeling but no intervention on the interoperation layer).

Since currently existing CSPs are heterogeneous in terms of their properties and extensibility, different CSPs have different degrees of capabilities for their external interfaces. This makes it extremely difficult to find a general approach for the integration. To solve this problem, a CSP emulator (CSPE) is designed with a simple simulation engine. The CSPE is intended to emulate the functionality and interface to a CSP and can be used to investigate and to compare various interoperability solutions. Based on the CSPE, the requirements for integration of CSPs and the HLA are investigated and interfaces for the CSPI-PDG (COTS Simulation Package Interoperability Product Development Group) interoperability reference models Type I and Type II are implemented and tested.

Another important feature of this research comes from the synchronization approach. Compared to optimistic synchronization, the conservative approach is easy to implement. That is why almost all current work on integrating CSPs and the HLA is based on conservative synchronization. However, the optimistic approach can exploit parallelism and achieve promising performance in situations where causality errors may occur but in fact seldom occur. The optimistic approach is integrated into the generic interface using a middleware approach. It can handle the complex rollback procedure on behalf of CSPs and the simulation models and release much burden from them. In addition to introducing optimistic synchronization to improve performance, middleware is also developed to solve the shared state problem using conservative synchronization. It can remove the time constraint

imposed on the lower bound on timestamp calculation, which in turn will improve the time advancement rate.

Some case studies are developed to demonstrate the interoperability and investigate the time synchronization in distributed simulation composed of CSP-based simulation components. Two of them are designed using the CSPE: one is a semiconductor wafer manufacturing model, built by integrating two CSPE model components; the other is the integration of a flight component system running on the CSPE with a business process for the inventory management. Furthermore, a real CSP – IBM's WBI Modeler is experimentally modified to support distributed simulation based on the proposed generic architecture, and a demand conditioning process model is created to evaluate the HLA-compatible WBI Modeler.

ACKNOWLEDGEMENTS

Firstly, I would like to express my heartfelt appreciation to my supervisor, Dr. Stephen John Turner, for his remarkable and heuristic guidance throughout my Ph.D study. As a patient listener, a helpful advisor and a strong supporter, he always encourages me to explore my own ideas. In particular, I want to thank him for showing me some of the most valuable personal characteristics and right attitude.

I also would like to give my thanks to Dr. Simon Taylor from Brunel University. It is really fruitful and enjoyable to discuss my work with him as he has amazing skills in explaining abstract and complex concepts with simple and vivid examples. Great help also comes from Dr. Wei Junhu, Mr. Chen Dan and Dr. Cai Wentong from Parallel and Distributed Computing Center (PDCC), and Mr. Gan Boon Ping, Dr. Low Yoke Hean Malcolm, Ms. Chan Lai Peng and Dr. Peter Lendermann from Singapore Institute of Manufacturing Technology (SIMTech). Their generous sharing of experiences and suggestions always inspire me.

I am also greatly indebted to Dr. Zeng Yi, a warm-hearted labmate and an expert in programming. Without his help, I could not finish the implementation of my proposed architecture in the project.

Special thanks should be given to Dr. Steve Buckley, my supervisor during my internship in IBM T.J. Watson Research Center, USA in 2005, for his patience and kind guidance. In IBM I also received very good cooperation from Dr. Young M. Lee, Dr. Shubir Kapoor, Dr. David L. Jensen and Mr. Young-Jun Yoon, who instructed me a lot during my internship.

Furthermore I would like to thank Zhang Jing and Heru Adi Kusuma, two undergraduate students from Nanyang Technological University. Their final year projects helped to verify part of my work.

Thanks should also go to Ng-Goh Siew Lai, Lau Li Jun and Ek Ming Hong, the technicians of PDCC, for having maintained a wonderful work environment for my research.

At last, I would also thank my husband, my parents and all my friends who support me consistently.

LIST OF FIGURES

LIST OF TABLES

INDEX OF ABREVIATIONS

ALSP	Aggregate Level Simulation Protocol
API	Application Programming Interface
APS	Advanced Planning Systems
COTS	Commercial Off-The-Shelf
CSP	COTS Simulation Package
CSPE	COTS Simulation Package Emulator
CSPI-PDG	COTS Simulation Package Interoperability – Product Development Group
DDM	Data Distribution Management
DIS	Distributed Interactive Simulation
DLL	Dynamic Link Library
DMSO	Defense Modeling and Simulation Office
DoD	Department of Defense (USA)
DVE	Distributed Virtual Environment
EEP	External Entry Point
FDD	FOM Documentation Data
FED	Federation Execution Data
FOM	Federation Object Model
FQR	Flush Queue Request
GTW	Georgia Tech Time Warp Operating System
GVT	Global Virtual Time
HLA	High Level Architecture
LBTS	Lowest Bound on Timestamp
LP	Logical Process
LRC	Local RTI Component
MOM	Management Object Model

MPI	Message Passing Interface
NER	Next Event Request
NERA	Next Event Request Available
NZL	Near Zero Lookahead
OMG	Object Management Group
OMT	Object Model Template
PADS	Parallel and Distributed Simulation
PDES	Parallel Discrete Event Simulation
RID	RTI Initialization Data
RTI	Runtime Infrastructure
RO	Receive Order
SISO	Simulation Interoperability Standards Organization
SOM	Simulation Object Model
SRML	Simulation Reference Markup Language
TAG	Time Advance Grant
TAR	Time Advance Request
TARA	Time Advance Request Available
TSO	Timestamp Order
WBI	Websphere Business Integration
XML	Extensible Markup Language

GLOSSARY

Causality — Dependency of one event on another event. An event e_2 is causally dependent on an event e_1, if e_1 has a smaller timestamp than e_2 and e_1 changes the state of the simulation model, e.g., a state variable, which influences e_2 directly or indirectly [STR01].

Conservative Synchronization — A time management mechanism that prevents a distributed simulation from processing messages out of timestamp order. This is in contrast to optimistic synchronization. A simulation using conservative synchronization only processes events when it is guaranteed that no future event with a smaller timestamp will be received.

COTS Simulation Package (CSP) — A kind of Commercial Off-The-Shelf product. The meaning of the term "commercial" is a product customarily used for general purposes and which has been sold, leased, or licensed to the general public. As for the term "off-the-shelf", it can mean that the item is not developed by the user, but already exists [MOT02].

Distributed Virtual Environment (DVE) — A DVE is a system that enables many geographically distributed participants to communicate and interact with each other in a three-dimensional, high-fidelity virtual world [ZCL04].

Event — An event is an abstraction used in the simulation to model some instantaneous action in the physical system.

Event Granularity — In this thesis, event granularity is defined as the computation time to process a task.

Federate — A component of a simulation system that has a single point of attachment to the RTI. A federate may be composed of one or many independent processes running on one or many hosts; from the perspective of the RTI, a federate is a unit. A federate interacts during execution with other federates through the RTI. Federates may also interact with each other directly as long as no FOM related data is involved.

Federation — The alliance of one or more federates acting together in a distributed simulation to achieve a certain objective is called a federation. A federation can be seen as a contract among the participants about each one's responsibilities in a common distributed simulation.

Interoperability	The ability of two or more systems or components to exchange information and to use the information that has been exchanged [IEEE 90].
Logical Process	A term from the area of parallel simulation for a submodel which cooperates with other submodels in a common distributed simulation. It usually refers to submodels in discrete event simulations.
Lookahead	Lookahead is a guarantee from a federate (or logical process) that it will not generate any messages with a timestamp smaller than its current time plus the value of lookahead. If a federate's current time is T, and its lookahead is L, any message generated by the federate must have a timestamp of at least T+L. Lookahead plays an important role in conservative synchronization.
Hardware-in-the-loop	A term for a simulation model, in which a real subsystem operates together with a simulated subsystem (usually in real-time).
Human-in-the-loop	A term for a simulation model, in which a real person is an integral part of the simulation model. The model receives input from the person in real-time and reacts to the input accordingly.
Optimistic Synchronization	A time management mechanism that allows causality errors to occur, but which can detect them and recover to a previous safe state. This is in contrast to conservative synchronization. A simulation using optimistic synchronization assumes all the events are safe and processes them optimistically although future events with smaller timestamps may be received later.
Reusability	The degree to which a software module or other work product can be used in more than one computing program or software system [IEEE 90].
Simulation	The technique of imitating the behaviour of some situation or process (whether economic, military, mechanical, etc.) by means of a suitably analogous situation or apparatus, especially for the purpose of study or personnel training [Oxford English Dictionary].
Simulation System	A simulation system includes a simulation model as well as the underlying simulation tool (mostly commercially) developed to facilitate the creation of the simulation model.
Simulation Time	Simulation time is the abstraction of physical time.

CHAPTER

1

INTRODUCTION

1.1 Background

1.1.1 Simulation and Parallel / Distributed Simulation

With the rapid development of computer technology, it is becoming increasingly possible to recreate real-world scenarios in a manner which is useful, safe, and affordable. This type of computer-based recreation of real scenarios is commonly referred to as *computer simulation.* Simply stated, computer simulation is a system (model) that represents or emulates the behavior of another system (physical system) dynamically.

In terms of state, computer based simulations are typically divided into two fundamental types, referred to as continuous and discrete. While a continuous simulation changes its state continuously over time, a discrete simulation only changes its state at discrete points of time when an event[1] occurs. In discrete simulation, the events are arranged with a timestamp and executed in increasing order to keep causality, in other words, to ensure the future cannot affect the past. Discrete simulation can further be classified by the way simulation time[2] is progressed. In time driven discrete simulation, the simulation time is advanced in time steps of constant size. For general-purpose simulation, this approach is not efficient because of the difficulty of selecting a proper constant step size, especially when events are irregularly dispersed over time [FER95]. However, this can be avoided in event driven discrete simulation where time is advanced according to the next earliest

[1] An event is an abstraction used in the simulation to model some instantaneous action in the physical system.

[2] Simulation time is the abstraction of physical time.

occurring event. The bulk of this thesis deals with the latter approach named discrete event simulation (DES).

As the nature of the simulations becomes more complicated, the performance of the simulation is limited by the performance of the computer on which it is running. However, this problem can be partially circumvented by utilizing a parallel computer architecture or large computer networks to spread the processing load among computers. In this way, large environments can be represented in a *parallel and distributed simulation* (PADS). Conceptually, parallel and distributed simulation is defined as a technology that enables a simulation program to be executed on a computing system containing multiple processors with shared memory or interconnected by a communication network.

Generally, parallel simulation and distributed simulation are mainly distinguished in the following way: parallel simulations execute on a set of computers confined to a single cabinet or machine room, while distributed simulations execute on machines that are geographically distributed across a building, university campus, or even the world [FUJ00]. As opposed to parallel simulation which is mostly concerned with execution speedup, distributed simulation can also facilitate hybrid system modeling that is even geographically dispersed. Complex hybrid system modeling may require distributed simulation due to system complexity, performance and interoperability requirements, and so on. Each simulation component should interoperate with other components, possibly created using different tools. This could be achieved by using some Modeling and Simulation (M&S) standard. This would allow the creation of distributed simulations, where the components run on different machines and different platforms [STR99].

1.1.2 Synchronization Issues in Distributed Simulation

In simulation, the physical system is often viewed as being composed of some number of interacting physical processes. Each physical process is modeled by a *logical process* (LP). So the whole distributed simulation could be considered as a collection of LPs communicating by exchanging timestamped messages. In a sequential simulation, by using a centralized list of pending events [FUJ00], event processing in timestamp order can be easily accomplished. However, it is not the same case in parallel/distributed simulation. Causality errors may occur resulting from out-of-order event processing since concurrent executions often happen between different LPs. The goal of the synchronization mechanism is to ensure that each LP processes events in increasing timestamp order, that is it obeys the *local causality constraint* [FUJ01]. Violating this constraint will incur anomalous behavior in that the future can affect the past. So it is very important to adopt effective synchronization algorithms to coordinate the executions between the LPs. In general, current synchronization algorithms can be divided into either conservative or optimistic approaches. The

conservative approaches take precautions to avoid violating the local causality constraint. In contrast, optimistic approaches allow violations to occur, but are able to detect and recover from them. Some famous algorithms will be discussed in Chapter 2.

In recent years, distributed simulation has become increasingly important as a strategic technology for linking simulation components of various types at geographically different locations [FUJ00]. For example, with the globalization of markets, the effective practice of supply chain management is crucial for improving a company's competitive advantage, but a typical supply chain simulation may comprise of companies from various enterprises, in a number of countries [TCG01]. Each of these companies may have its own simulation program and it is ideal to reuse these existing models. In addition, the existing different models may be implemented using different languages / simulation packages and on different hardware platforms. Thus, there is a pressing need for a common standard that meets such reusability and interoperability requirements in distributed simulation.

The High Level Architecture (HLA) [IEEEP1516][IEEEP1516.1][IEEEP1516.2][IEEEP1516.3] has recently been accepted as an IEEE standard for distributed simulation. Its chief intent is to allow simulation applications to be created by combining a number of simulation components [FUJ98]. The HLA focuses on interoperability and reusability of the components (called federates) and offers time management interoperability which allows simulations using different internal time management mechanisms to be combined in a single federation (collection of federates) execution. That means transparency of time management mechanisms between components can be achieved. In this thesis, the benefits of the HLA will be investigated and utilized, and HLA-based simulations composed of multiple simulation components will be implemented.

1.1.3 Commercial Off-The-Shelf (COTS) Simulation Packages

In the past, simulations were often developed using some programming languages such as C++ or Ada. However, constructing simulations in a conventional language has a number of disadvantages and imposes a significant burden for the developers [SSK98]:

- A general purpose language is more difficult to learn.
- It is too easy to make mistakes which have disastrous consequences, but which are difficult to find, e.g., faulty use of pointers.
- Such languages often lack an inherent mechanism for describing parallelism.
- The debugging tools are simulation unaware, e.g., they operate at a level far below that which would be convenient for most simulationists (modelers who develop the simulation model).

To meet the pressing demand for an easy and safe way to implement simulation modeling,

Commercial Off-The-Shelf Simulation Packages (referred to in this thesis as CSPs) have been developed. A CSP is defined as being one that supports the creation of models using some form of visual interactive modeling interface [TAY01]. CSPs from different vendors support simulation objects that cover a wide variety of application fields, such as business processes (e.g., Arena [ARENA]), manufacturing (e.g., Flexsim [FLEX]), health care (e.g., GPSS/H [HEC95]), supply chain analysis (e.g. Pro Model [PMODEL]), and transportation (e.g., SLX [HEN97]). More comparison and analysis of current popular CSPs can be found in [SWA03].

Using CSPs to build systems has been proposed as a means of developing software with reduced risk and cost while increasing the functionality and capability of the system. Building a system based on CSPs involves buying a set of pre-existing, proven software components, building extensions to satisfy local requirements, and gluing the software components together. The advantages claimed are that the CSPs are honed in the competitive marketplace resulting in improved capability, reliability, and functionality for the end user.

1.2 Motivation for Project

1.2.1 CSP Interoperability

The main motivation for this project comes from the demand for interoperability between multiple simulation components in a distributed environment. Basically, the development of distributed simulations requires: (1) a simulation language or package to construct the model components, and (2) tools to implement a protocol by which model components can be interconnected [SSK98]. It might be expected that multiple types of functionality would be integrated into a single simulation package which one could purchase “off the shelf”. However, most current CSPs are specially designed for certain purposes and market niches. Moreover, some large scale simulations can only be modeled using a set of model components working together, instead of working on a single computer. Other reasons such as the reusability of existing models also motivate the need for interoperability of the CSPs.

It is desirable to develop a distributed simulation based on CSPs by customizing the various simulation packages to suit local requirements, and gluing the components together. This could come close to a plug-and-play scenario for building models by plugging together individual components. For example, wafer fabs are currently moving towards 300-mm wafer production to replace 200-mm wafers which are currently used in the semiconductor industry. These fabs are highly automated with minimal human involvement in the loading, movement, and unloading of wafer lots. Efficient scheduling of the material handling process is thus critical to achieve maximum tools efficiency.

Generally, one CSP (AutoMod [ROH03]) is used for the material handling simulation, while another CSP (AutoSched AP [ASAP]) is used for the manufacturing process simulation [GTT05]. Therefore, the interoperability of AutoMod and AutoSched AP will be an important issue for integrating the material handling and manufacturing process models. Another example of the need for distributed simulation is the "Borderless Fab" model [LEN06]. Many semiconductor manufacturers may have several fabs to produce a given device. The borderless fab concept was introduced to share capacity across multiple fabs, especially in close proximity to each other, to achieve effective resource utilization and shorter cycle times of a system. It is feasible to study the borderless fab application scenarios by connecting the simulation models of individual fabs and executing the simulation in a distributed manner.

While there are various standards for distributed applications, such as the Message Passing Interface (MPI) [MPI], Distributed Interactive Simulation (DIS) protocols [DIS94] and the Aggregate Level Simulation Protocol (ALSP) [WSW91], HLA is chosen as the communication broker for systems built by different CSPs. The reasons are due to the drawbacks of other standards as compared with the HLA. Although MPI applications are very portable, it cannot support true interoperability among different platforms and cannot achieve language interoperability for application development [GKP96]. DIS and ALSP are the predecessors of the HLA and have a limited domain. A detailed comparison of HLA with similar techniques in [STR01] reveals that the HLA is the most advanced technology for interoperability among simulation components currently available. Furthermore, the HLA is applicable not only to the real-time training community but also to the discrete event community and to virtual and live simulations.

With the wide use of CSPs and the advent of the High Level Architecture (HLA) for Modeling and Simulation, HLA based CSPs become an appropriate choice for simulation developers. In recent years, it has become a hot topic to which many researchers are dedicated.

1.2.2 CSPI – PDG Standards

The advent of the HLA offers the basis for the integration of general heterogeneous distributed simulation components using CSPs. However, the HLA is not a complete solution to CSP interoperability since it only focuses on the infrastructure issues. It is difficult to match model information represented in the CSPs to the object/interaction concept in the HLA standard. In addition, the terminology between different CSPs differs as there is no internationally recognized naming convention. Another difficulty arises from the complexity of the HLA itself and the time-consuming work to integrate the CSP with the HLA. In addition, the lack of expertise in distributed simulation (e.g. time synchronization algorithms) may also be a barrier to the development of HLA-

based distributed simulation. There is a pressing need for a common standard for interoperability of CSPs.

In 2005, the Commercial Off-The-Shelf Simulation Package Interoperability – Product Development Group (CSPI-PDG) was approved by the Simulation Interoperability Standards Organization (SISO). Previously known as the HLA CSPI Forum [HLA-CSPI], it is dedicated to creating a standardized approach to support the interoperation of discrete event models created in CSPs using the IEEE 1516 High Level Architecture. The following are the planned suite of CSP interoperability standards:

- Interoperability Reference Models (IRMs)
  - IRM Type I: Asynchronous Entity Passing (Basic)
  - IRM Type II: Synchronous Entity Passing (Bounded Buffer)
  - IRM Type III: Shared Resources
  - IRM Type IV: Shared Events
  - IRM Type V: Shared Data Structures
  - IRM Type VI: Shared Conveyors
- Interoperability Frameworks (IF)
- COTS Simulation Package Emulators (CSPE)
- Entity Transfer Specification (ETS)

The *Interoperability Reference Models* (IRMs) outline different integration needs of CSP interoperability. The Type I IRM asynchronous entity passing deals with the common requirement of transferring entities between simulation models. In the Type II IRM synchronous entity passing, the sending model needs to transfer entities to a bounded queue or a workstation with limited capacity in some remote model. The entities can be transferred only when the destination side is not full (bounded queue) or not blocked (workstation with limited capacity). The other four types of IRM deal with the sharing of resources, events, data structures and transportation tools across simulation models. The *Interoperability Frameworks* (Ifs) define the HLA-based solution to each IRM. A *CSP Emulator* (CSPE) is intended to emulate the functionality and interface to a CSP and can be used to investigate and to compare various interoperability approaches. The *Entity Transfer Specification* (ETS) defines appropriate data exchange representations to specify the data exchanged in an IF. This thesis focuses on identifying and solving the problems for the first two IRMs and the corresponding experiments are carried out using a CSPE developed in this research.

1.2.3 Zero or Near Zero Lookahead in Simulation Models

In addition to the standardized interface to integrate the CSP to the HLA, another very important issue in distributed simulation is the time management mechanism. The most common approaches

for time management in the HLA are time stepped, event driven (conservative) and optimistic [VAM99]. Some systems use time stepped approaches, which request the advancement of simulation time by fixed time intervals. There are some disadvantages of this approach, e.g., performance overhead and inaccuracies.

For synchronizing discrete event simulations, conservative synchronization schemes are often preferred as they are easy to implement. However, the Achilles' heel for the conservative approach is the need for *lookahead* to achieve good performance [FUJ00]. Lookahead is a guarantee from a federate (or logical process) that it will not generate any message with a timestamp smaller than its current time plus the value of lookahead. If the guarantee is zero, then the conservative protocols perform poorly.

Therefore, exploiting lookahead from simulation models is critical to achieve good performance and a number of techniques [NIC88] [CAT90] [COS90] [MEB99] [DBS01] have been proposed to maximize lookahead. However, there are many situations where zero lookahead cannot be avoided. For example, in a system in which real and simulated components are integrated, the lookahead value for the links that connect the real components to the simulated components must usually be set to zero. It is because the future behavior of the real system is hard to predict, e.g., the breakdown of a machine, the latency of the network, or any unexpected event in the real world. All this unpredictability results in no knowledge about the time of the next message received directly from the real components.

In some other cases, even for a collection of pure simulation components, lookahead may have to be set to zero. In general, lookahead is chosen by considering all the timestamp order (TSO) messages sent by a federate. The smallest time increment required by a TSO message is chosen as the simulation lookahead. In some simulation models where there exist some shared variables (e.g., CSPI-PDG Type III and Type V IRM), this could mean zero lookahead, because the shared variables have to be written and read at the same instant of simulation time. With non-zero lookahead, new updated information for a shared variable at simulation time T could not be sent out until its time reaches T plus its lookahead.

Furthermore, the actual calculation of the value of lookahead may be problematic. For instance, if the distribution governing the timestamps of event messages is stochastic, then there is no guarantee that zero might be sampled from this distribution; so there is no guarantee that lookahead could not be zero [SRT01].

Moreover, near zero lookahead may be generated in some types of models. An obvious case is CSPI-PDG Type II IRM, where some bounded queue or workstation with limited capacity exists in the model. The status of the queue or workstation needs to be updated dynamically to block or enable the

new entity to be transferred into the receiving model. The status information is sent in the form of status event with a small increment of the current simulation time since the status event is dependent on the entity sending event. Such a small increment leads to the near zero lookahead value in the whole simulation.

Above all, it is worth considering optimistic approaches since optimistic approaches can be free of the constraints from lookahead.

1.3 Objectives

The contributions of the project described in this thesis are fully embodied in the following objectives: a generic architecture for integrating CSPs with the HLA, contribution to CSPI-PDG standards, optimistic synchronization approaches and solutions for shared state in conservative synchronization approaches.

1.3.1 A Generic Architecture for Integrating CSPs with the HLA

As discussed before, a component-based approach for building complex simulation models with heterogeneous CSPs is missing. Such an approach should support three aspects in the construction of complex simulation models: (1) multiple submodels developed with best-suited simulation packages; (2) reuse of some existing models by re-combining them and; (3) integration of physically distributed models. Since the High Level Architecture (HLA) [KWD99] provides features of interoperability and reusability, it is desirable to introduce an HLA interface for CSPs. However, the diverse characteristics of different CSPs and the complexity of the HLA standard itself will be a barrier to widespread adoption of the HLA in CSPs. One objective of this project is to design a generic architecture for integrating CSPs with the HLA and adopt a nearly implicit approach for the modelers (those who develop the models using the CSP). That means (1) the generic architecture should provide a generic interface which is independent of the platforms and the CSP used to build the model; (2) all HLA functionality should be hidden from the modeler since the CSP and its underlying software handle all the HLA synchronization and communication. In this way, the modelers can design their model components in a “plug & play” manner (involving only process modeling but no intervention due to the needs of interoperability).

1.3.2 Contribution to CSPI-PDG Standards

Another objective of this project is to provide a significant contribution to CSPI-PDG standards.

Based on the CSPI-PDG Type I and Type II IRMs, the requirements for the integration of CSPs and the HLA are investigated and an interface is proposed. The CSP integrates with the HLA through a generic interface called DSManager. The interface consists of a set of functions to be invoked by the CSP when necessary. The C++ / Java based HLA services are wrapped by "normal" C functions, that can easily be integrated with most of the current CSPs written in C, C++, Java or VB. Another important feature of the DSManager is to try to hide the HLA concept from both the CSP and the model. It is difficult to match model information represented in the CSPs to the object/interaction concept in the HLA standard. In addition, the terminology between different CSPs differs as there is no internationally recognized naming convention. The interface adopts a generic approach based on the concept of entity transfer. For CSPI-PDG Type I IRM, the interface focuses on the general problem of entity transfer specification as well as providing the basic functions necessary for the whole simulation life cycle. For CSPI-PDG Type II IRM, the entities can only be transferred when the sending model make sure there is space available in receiving side. So the interface also supports the status updating for the receiving model and the status checking in the sending model.

In addition, another part of the CSPI-PDG standards, a CSP Emulator (CSPE), is also designed in this project. The CSPE is intended to emulate the functionality and interface to a CSP. It supports the creation of either a standalone model or a model component that is part of a distributed simulation. The CSPE can emulate the functionality and interface to a CSP and can be used to investigate and to benchmark alternative interoperability solutions. Another benefit of the CSPE is to provide a suggestion as to how current CSPs may add an HLA capability to support distributed simulation. In the thesis, some case studies are performed using the CSPE.

1.3.3 Optimistic Synchronization Approaches

As discussed before, conservative synchronization algorithms depend heavily on a lookahead of reasonable size, which is not guaranteed in many practical application models. In addition, conservative approaches cannot fully exploit the parallelism in the application because they must guard against a worst-case scenario, which may seldom actually occur in practice. So it is worth considering optimistic approaches. Although the performance overhead caused by state saving and the rollback operation can be high, it still has some advantages over other approaches in some certain areas. For example, in the absence of lookahead or in a simulation system where topology information among logical processes changes during the execution, the optimistic approach seems more effective. Another advantage of this approach is that it allows the exploitation of parallelism in situations where causality errors might occur, but where in fact they do not occur [STR01].

Little work has been done on HLA-based optimistic simulation. Even though the HLA offers some

services for the optimistic approach, a lot of work including state saving and recovery is still required to build optimistic federates. To relieve the modeler from the burden of handling problems in the optimistic approach, middleware is introduced, named rollback controller, to handle most of the complicated rollback procedure on behalf of the CSP and the simulation model. The simulation modeler can still build their simulation federate as before and link the middleware library without worrying about the underlying implementation details.

1.3.4 Solutions to Shared State in Conservative Synchronization

As previously discussed, shared state will incur the problem of zero lookahead as the state variables have to be written and read at the same instant of simulation time. When conservative synchronization is applied for the simulation, it is difficult to gain good performance. In [MEH93] two methods are suggested to implement shared variables for a conservative protocol. The first approach using query events is sometimes referred to as "pull processing" because each federate is responsible for "pulling" the information it needs. The second approach called "push processing" is where the federate automatically "pushes" the value of the required state information to other federates whenever the variable changes.

Based on these two processing methods and the HLA/RTI APIs, two algorithms, namely *pullRO* and *pushRO* are proposed. In *pullRO* or *pushRO*, some of the timestamp order (TSO) messages that cause zero lookahead values are replaced with receive order (RO) messages. This removes the time constraint that these messages impose on the calculation of the simulation time, which in turn will improve the time advancement rate of federates. To free the users (simulation developers) from the complex implementation, a middleware approach is introduced to extend the low-level services provided by the HLA/RTI. Meanwhile, the original semantics of the RTI APIs still remain for the easy use by the simulation developers.

1.4 Structure of the Thesis

The thesis is organized into eight chapters.

Chapter 2 provides an overview of related work in parallel and distributed simulation and CSPs. It reviews different time management algorithms, gives an introduction to the High Level Architecture (HLA), the characteristics of current popular CSPs, and discusses the integration of CSPs with the HLA. In addition, this chapter also reviews the six types of CSPI-PDG interoperability reference models.

Chapter 3 proposes a generic architecture to facilitate the integration of a CSP with the HLA in a way that is transparent to the user. It is realized by introducing a middleware named DSManager between the CSP and the HLA Runtime Infrastructure (RTI). Both the DSManager for Type I and Type II IRMs are designed and discussed.

Chapter 4 describes a CSP Emulator (CSPE) which is intended to emulate the functionality and interface to a CSP and can be used to investigate and compare various interoperability approaches. In addition to providing support for standalone models as provided by current CSPs, the CSPE has some new features needed for building distributed models. Using typical models of CSPI-PDG Type I and Type II IRMs, experiments are conducted to evaluate the generic architecture, in particular: 1) to evaluate the correctness of the CSPE and the interface for Type I and Type II IRMs; 2) to compare the performance between standalone and distributed simulation varying experimental factors such as the lookahead value and event granularity[3].

Chapter 5 introduces the design and implementation of a generic rollback manager for the optimistic approach using the HLA. It can improve the performance for some Type II entity passing models since lookahead is difficult to exploit in such models. To release the modelers from the burden of the complicated rollback procedure, the middleware is implemented in an almost transparent way. Further, experimental results are also shown to compare the performance between conservative and optimistic synchronization approaches using a typical Type II IRM.

Chapter 6 describes two new algorithms using conservative synchronization to eliminate the zero lookahead constraint introduced by shared state in the simulation. The design and implementation of these two algorithms using the HLA is presented. This will contribute to type III, type V, and type VI IRMs as there are similar problems existing in these types of reference models.

Chapter 7 presents some case studies for the proposed generic architecture and middleware in this research. In addition to two case models (a semiconductor wafer manufacturing model and a business application) built using CSPE, a demand conditioning process model is tested using a real CSP – IBM's WBI Modeler, which is linked with the DSManager. The modification to the WBI Modeler to support the HLA will be discussed.

Chapter 8 summarizes the achievements of this research and discusses future work in this research area.

[3] In this thesis, event granularity is defined as the computation time to process a task.

CHAPTER

2

LITERATURE REVIEW

2.1 Parallel and Distributed Simulation

To accelerate the execution of simulations, parallel and distributed simulation technology[4] has been developed to enable a simulation program to be executed on parallel / distributed systems, namely systems composed of multiple interconnected computers. While parallel simulation is executed on multiple processors (CPUs) with low communication latency, distributed simulation is executed on loosely coupled systems where interactions take much more time. Fujimoto [FUJ00] summarized the principal benefits to executing a simulation program across multiple computers as follows: 1) Reduced execution time; 2) Geographical distribution; 3) Integrating simulators that execute on machines from different manufacturers; 4) Fault tolerance. These advantages as well as other reasons, such as scalability and group working, motivate the use and development of parallel and distributed simulation.

2.1.1 Time and Logical Processes

Since a simulation is a system that represents or emulates the behavior of a physical system over time, it may be confusing that what kind of time is meant when discussing parallel and distributed simulation. Basically, there are three different types of time in parallel and distributed simulation. *Physical time* refers to time in the physical system. *Simulation time* is an abstraction used by the simulation to model physical time (a precise definition will be given in Definition 2.1.1) and it is also called virtual time or logical time. *Wallclock time* refers to the time during the execution of the simulation program [FUJ98].

[4] In this thesis, parallel and distributed simulation refers to parallel and distributed discrete event simulation.

Definition 2.1.1 Simulation time. "Simulation time is defined as a totally ordered set of values where each value represents an instant of time in the physical system being modeled. Furthermore, for any two values of simulation time, T_1 representing physical time P_1, and T_2 representing P_2, if $T_1 < T_2$, then P_1 occurs before P_2, and $(T_2 - T_1)$ is equal to $(P_2 - P_1)*K$ for some constant K. If $T_1 < T_2$, then T_1 is said to occur before T_2, and T_2 is said to occur after T_1." [FUJ00]

Some simulations are referred to as as-fast-as-possible simulations because their goal is to complete the execution of the simulation as quickly as possible. Typically analytic simulations belong to this category. In some other simulations when human or physical devices are involved, advances in simulation time are paced to be in synchrony with advances in wallclock time. Such simulation executions are often referred to as real-time executions, and simulators operating in this mode are called real-time simulators. Real time simulations are mainly applied in Distributed Virtual Environments (DVEs)[5] as *human-in-the-loop* simulations or *hardware-in-the-loop* simulations.

In parallel and distributed simulation, the overall simulation is divided into a set of inter-communicating *logical processes* (LPs). These LPs interact by exchanging timestamped messages or events. Each LP can be viewed as a sequential discrete event simulation which maintains some local state and a Future Event List (FEL) which contains a list of timestamped events that have been scheduled for this LP. By processing these events, the LP may change its state variables and schedule additional events[6] for itself and other LPs. The new events generated for other LPs will be transferred to the destination LPs. Each LP maintains a simulation time clock that indicates the timestamp of the most recent event processed by the LP. The description of a logical process is shown in Figure 2.1.1.

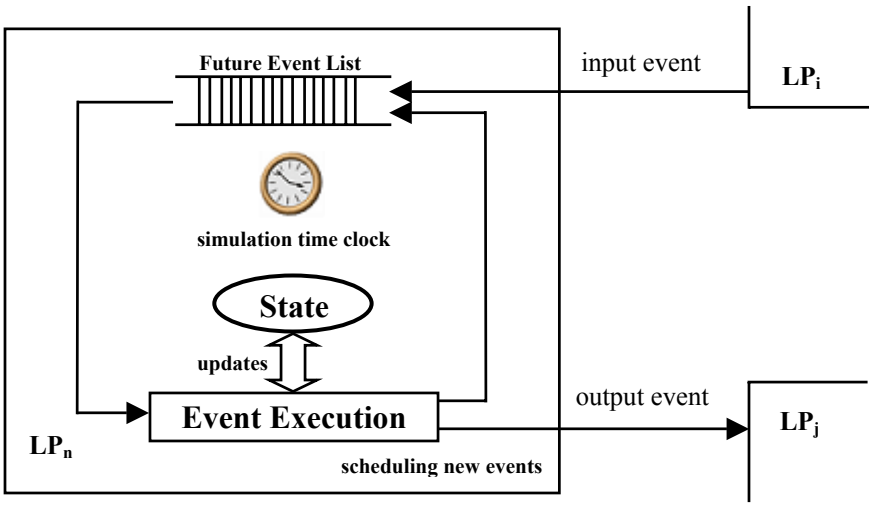


Figure 2.1.1: An LP in parallel and distributed discrete event simulation

[5] "A DVE is a system that enables many geographically distributed participants to communicate and interact with each other in a three-dimensional, high-fidelity virtual world." [ZCL04]
[6] The mechanism for creating a new event in the simulation is called scheduling an event.

To ensure these LPs produce exactly the same results as the corresponding sequential simulation where events are processed according to the timestamp order, the local causality constraint should be followed.

Definition 2.1.1.2 Local Causality Constraint. "A parallel simulation obeys the local causality constraint if and only if each LP processes events in nondecreasing timestamp order (TSO)." [MIS86]

It is important to note that the local causality constraint is a *sufficient condition* but not a *necessary condition* [FUJ01]. It is because two events occurring in the same LP may be concurrent (independent of each other) and can be processed in any order. Many approaches have been developed to synchronize asynchronous LPs based on obeying the local causality constraint. Broadly they fall into two families: conservative synchronization algorithms and optimistic synchronization algorithms. Detailed discussions will be shown in the following sections.

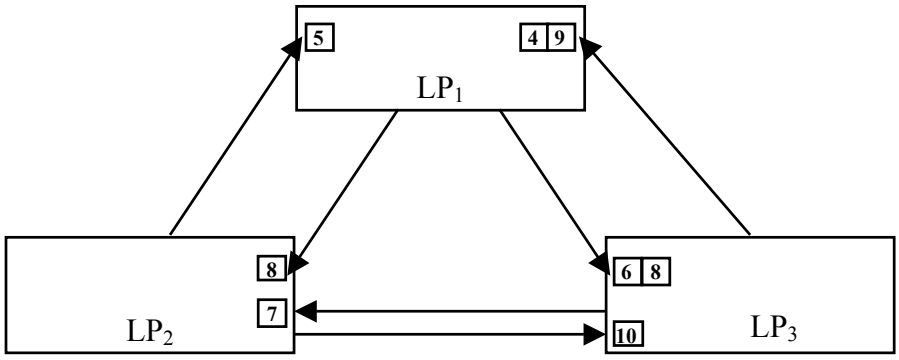
2.1.2 Conservative Synchronization Approaches

The key feature of conservative synchronization approaches is that any LP only processes *safe events* from its *Future Event List* (FEL) to keep the local causality constraint. An event is considered as a safe event when there is guarantee that no other event with a smaller timestamp will arrive. This may lead to a deadlock situation if an LP does not have any safe event to process.

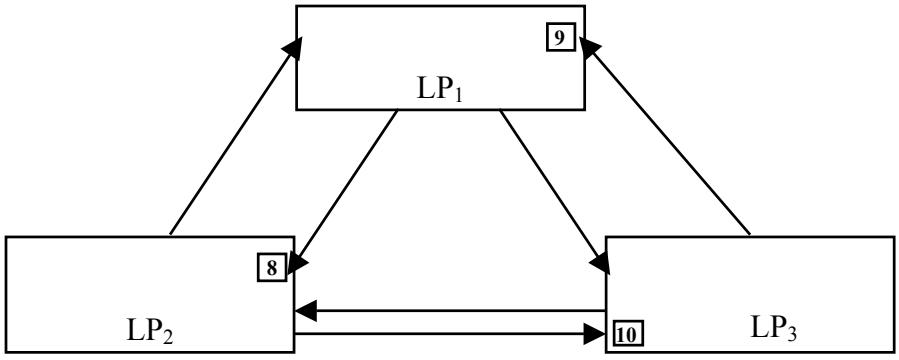
2.1.2.1 Deadlock Situation

When conservative synchronization approaches are applied, the interacting LPs in the distributed simulation must move forward carefully to guarantee no causality violation will happen. This, however, may incur a deadlock situation. Consider the Chandy/Misra/Bryant (CMB) algorithm [BRY77], the original algorithm for the conservative synchronization approach. A distributed simulation is composed of a number of interacting *logical processes* (LPs). Each physical process is mapped to an LP in the simulation system and messages are mapped to events. Each LP has an incoming link for each other LP with which it could exchange messages. Messages arriving on each incoming link are stored in a first-in-first-out (FIFO) input queue, which is also in nondecreasing timestamp order. As long as an LP has an event on each incoming link, it is safe to execute the event with the smallest timestamp. If one of the links becomes empty, the LP will block until a new event is added to this incoming link. It should be observed that the algorithm, as described, is prone to deadlock, because a loop of processes may exist such that each process is blocked waiting for a messages from its predecessor in the loop.

Let us consider the following situation in Figure 2.1.2 (a). Each LP has an input queue holding the incoming messages from each of the other LPs. Suppose the minimum amount of travel time to send an event from one LP to another is 5 time units. In LP_1, the input queue for LP_2 has 1 message with timestamp 5 and the input queue for LP_3 has messages with timestamp 4 and 9. So LP_1 will now process safe messages in this order: 4(LP_3) and 5(LP_2). Assuming that no new messages have been received, LP_1 will block at this point because of the possibility that a new message will arrive later from LP_2 with a timestamp smaller than 9. The same thing may happen at LP_2 and LP_3. LP_2 blocks waiting for new messages from LP_3 while LP_3 blocks waiting for new messages from LP_1. As shown in Figure 2.1.2 (b), each LP is waiting on an incoming link for the smallest link timestamp value because the corresponding queue is empty. All the three LPs are blocked, even though there are events in other input queues that are waiting to be processed.



(a)



(b)

Figure 2.1.2: Deadlock situation in the conservative synchronization approach

2.1.2.2 Different Algorithms for the Conservative Approach

Many algorithms have been proposed to seek an efficient way to overcome this specific deadlock problem in the conservative approach. Before giving a general description, the definition of

lookahead, which is the key ingredient for all conservative synchronization methods, is given.

Definition 2.2.1 Lookahead. "If a logical process at simulation time *T* can only schedule new events with timestamp of at least *T*+*L*, then *L* is referred to as the lookahead for the logical process." [FUJ00]

In general, most of the conservative approaches can be categorized into these types: deadlock avoidance method, deadlock avoidance method with lookahead optimization, deadlock detection and recovery method and conservative time windows method. A brief summary is provided in the Table 2.1.1:

Table 2.1.1: Summary of conservative synchronization approaches

Method	Description	Features	Representation
Deadlock avoidance	Whenever an LP finishes processing an event, it sends a null message to other LPs with which it is linked. The null message with timestamp T_{null} can be treated as a "guarantee" that this LP will not send a message with a timestamp smaller than T_{null}.	• Deadlock Avoidance • A large number of null messages can slow down the simulation.	CMB Protocol (null message protocol). [BRY77][CDM79][MIS86]
			Demand-Driven protocol [NIR84][MIS86][BAS88] LP will request the next message when it has no safe event to process.
Deadlock avoidance with Lookahead Optimization	Lookahead is the ability to predict the future. With larger lookahead, the LP could provide a better "guarantee" that this LP will not send a message with a timestamp smaller than its current time plus lookahead.	• It allows more events to be safely processed. • The performance of the null message algorithm depends critically on the lookahead value.	Precomputing the service times for some future events. [NIC88]
			Carrier Null Messages protocol [CAT90] • Null message carries additional information on lookahead and the route taken.
			Automatic Lookahead Computation. [COS90] • Using a control flow graph the dependencies of each output can be discovered and a maximum value for each output null message can be calculated.
			Path Lookahead. [MEB99] • Views simulation model as data flows rather than a collection of objects and lookahead is computed for each data flow path.
			Improves lookahead based on compiler analysis. [DBS01]
Deadlock Detection and Recovery	Instead of avoiding deadlocks, a simulation is allowed to enter a deadlock. The deadlock can be broken by observing that the messages containing the smallest timestamp in the entire simulation are always safe to process	• It avoids the null message traffic as well as allowing zero lookahead. • This method often results in sequential executions prior to a deadlock which would adversely affect the overall performance if the simulation model is prone to deadlock.	Deadlock Detection and Recovery. [MIS86]
			Local Deadlock Detection. [LIT90]

Synchronous Execution (Conservativ Time Windows)	It divides the execution of a parallel program into a sequence of steps, where each step involves a parallel computation that must be completed before execution moves on to the next step so that successive steps do not interfere with each other.	• It does not require a deadlock detection mechanism and allows the simulation execution determine which events are safe to process. • It will continually block and restart throughout the simulation.	Bounded Lag protocol. [LUB88] It limits how far one LP can lag behind another.
			YAWNs [NIC93] It creates windows small enough to guarantee that all processing within the window will be correct.
			Conservative Super-Step Algorithm [CLT97][LLG98a][LLT98][LLG99] • Computes a SafeTime and allows all the events with timestamp smaller than SafeTime to be processed.

2.1.3 Optimistic Synchronization Approaches

The key feature of optimistic synchronization approaches is that each LP "optimistically" assumes all events in its Future Event List are safe events and adheres to the local causality constraint by detecting causality errors and rolling back to a previous safe state. As the first and the most well-known optimistic synchronization algorithm, Jefferson's Time Warp mechanism includes the fundamental concepts and mechanisms such as rollback, anti-messages, and *Global Virtual Time* (GVT). This section describes the fundamental ideas introduced in Time Warp [JFD85] and an example execution system named Georgia Tech Time Warp (GTW) [FDP97] as well as the associated algorithms.

2.1.3.1 Checkpointing and Retraction

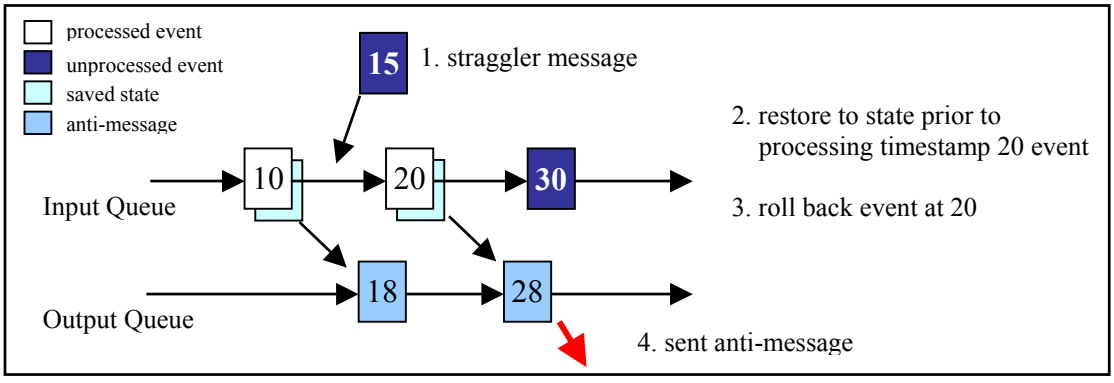


Figure 2.1.3: Rollback in optimistic synchronization approach

A Time Warp LP (TWLP) will repeatedly remove the smallest timestamped unprocessed event from the event list and process it. Unlike a conservative simulation executive, TWLP makes no check for safe events and blindly goes ahead and processes the next event. This may mean that a message in its

past is received. Such "late" arriving messages are referred to as *straggler messages*. At this moment, rollback occurs and the TWLP needs to do two operations: 1) restore to a previous safe state and reprocess the input messages; 2) cancel (retract) the events which have been sent to other TWLPs. Figure 2.1.3 gives a simple example for the rollback procedure.

For the first operation, each TWLP should have a checkpointing scheme that guarantees the ability to reconstruct state information related to past values of the simulated time. There are three main checkpointing techniques, named *copy*, *sparse* and *incremental*. In copy state saving [JES82][JFD85] a checkpoint of the state of the LP is taken prior to the execution of each event. Under sparse state saving [FUJ90][BEL92] the state of the LP is recorded periodically. Under incremental state saving [STE93][UCC93] a log recording changes to individual state variables is kept for each event computation. Based on these well-known schemes, some other new techniques have been developed, such as hybrid state saving [QUA97] and probabilistic checkpointing [QUA99].

For the second operation, each LP needs to keep an output queue. When rollback occurs, the LP performs retraction by sending an anti-message to annihilate the erroneous one. This anti-message is generated by examining the output queue of positive messages sent. When an LP receives an anti-message, it will check whether it has processed the corresponding positive message. If it has processed the positive message, this LP will also rollback and may have to send anti-messages to other LPs; otherwise, the positive message and anti-message will just simply annihilate each other.

Checkpointing and retraction will incur much memory overhead. A mechanism is required to reclaim unnecessary memory resources, which is called *fossil collection*. A concept referred to as Global Virtual Time (GTV) is proposed for this purpose.

Definition 2.1.3.1 Global Virtual Time. "Global Virtual Time at wallclock time T (GVT_T) during the execution of a Time Warp Simulation is defined as the minimum timestamp among all unprocessed and partially processed messages and anti-messages in the system at wallclock time T." [FUJ00]

GVT acts as a lower bound that guarantees the LP will not rollback to a previous state with a timestamp smaller than GVT. Thus, GVT defines a *commitment horizon* [JFD85]. Once GVT crosses the time of any event, that event can never be rolled back again and can therefore be committed. When an event is committed, an LP can release all the memory held by that event. Therefore, as GVT progresses, all but one of the saved states with timestamp smaller than GVT can be safely discarded. Similarly, input and output messages with a timestamp smaller than that of this state can also be discarded. GVT is useful in optimistic approaches not only for memory reclaiming but also for certain "irrecoverable" operations, such as I/O operations which cannot be easily undone. Many algorithms for efficiently computing or estimating GVT have been proposed, some of which can be

found in [SAM85] [BEL90] [MAT93] [XIC95] [CHT98].

2.1.3.2 Different Algorithms for the Optimistic Approach

Besides the rollback mechanism described above, some other advanced algorithms have been developed for optimizing performance in the optimistic approach. Table 2.1.2 shows and compares some well-known algorithms.

Table 2.1.2: Summary of optimistic synchronization approaches

Algorithm	Description	Advantages	Limitations
Lazy Cancellation [GAF88] [RFB90]	It is a technique that avoids canceling messages that are later recreated when the events are reprocessed. The process only sends an anti-message if the original message was not again created when the events were reprocessed.	• Avoids canceling messages recreated when rolled-back events are reprocessed; • Eliminates secondary rollbacks that occur in aggressive cancellation when an anti-message is unnecessarily canceled.	• Cancellation of incorrect computations is delayed. • Reprocessing events using lazy cancellation requires some additional overhead computation relative to aggressive cancellation to compare messages that are sent with anti-messages in the output queue. • It requires some additional memory to hold anti-messages during recomputation.
Direct Cancellation [FUJ89]	Use pointers to quickly send anti-messages.	• Eliminates the need for explicit anti-messages and an output queue; • Eliminates the need to search queues to locate events that must be cancelled.	Easily implemented in shared-memory machine, but complex for distributed memory architectures.
Local Rollback [DIR90]	When an LP sends a message, the process does not immediately send it until GVT advances beyond the send timestamp of the message.	• Cascaded rollbacks can be eliminated. • Avoids the need for anti-messages.	• Require some additional memory to store messages in buffer; • Delays the transmission of correct messages.
Lazy Reevaluation (Jump-forward optimization) [WES88]	If the state of the LP before re-processing the event after a rollback is the same as in the original execution of these events, the LP can skip re-processing the events.	It reduces the re-processing when straggler events do not affect the state of the logical process.	• Much cost of comparing state vectors, especially in the case that straggler events typically do modify state variables. • Comparison of state vectors is cumbersome and time-consuming when incremental state saving is used.

Wolf Calls [MAD88]	When a processor detects it has received a straggler message, it broadcasts a special control message that causes the processors to stop advancing until the error has been erased.	It can avoid "dog chasing its tail". (While rollback mechanism is canceling an incorrect computation, this incorrect computation is spreading throughout the simulation system.)	It may unnecessarily block some LPs.
Moving Time Window [SBW88] [RWJ89]	To set a bound on how far one logical process can advance ahead of others in simulation time.	It provides a simple, easy to implement mechanism to avoid "runaway LPs" from advancing far ahead of others.	• Cost of maintaining the window (frequent GVT computations or an efficient mechanism for estimating GVT). • The window does not distinguish correct computations from incorrect ones. • It is not immediately clear how the size of the window should be set, as it is dependent on the application.
Breathing Time Buckets [STE93]	To compute a quantity called the event horizon that determines the size of the time window.	It does not require anti-messages and therefore does not suffer from them.	Cost of maintaining the window, which requires frequent event horizon computations.
Lookahead-Based Blocking (Filtered Rollback) [LWS89]	Lookahead is used to determine which events are safe to process.	Reduce the number of rollbacks.	The value of lookahead determines the efficiency of this algorithm.
Transitive Dependency Tracking [DWG97]	An LP receiving a straggler broadcasts its rollback. On receiving this announcement, other LPs may rollback based on transitive dependency information.	• Eliminate the need for output queues. • Eliminate the problem of cascading rollbacks and echoing.	Cost of storing extra dependency information and increased processing time upon receiving a message.

2.1.4 Conservative vs. Optimistic Synchronization Algorithms

Table 2.1.3 summarizes some important distinctions among conservative and optimistic approaches. It shows that each kind of approach has its own advantages and drawbacks. That is why, after years of research in synchronization protocols, there is a no final conclusion on which one is the winner for all applications [FUJ00]. The fact remains that the optimal protocol for any particular situation is application dependent. A comparison of performance between CMB and Time Warp, and variations of the conservative and optimistic synchronization approaches, can also be found in [FJT01].

Table 2.1.3: Comparison between conservative and optimistic approaches

	Conservative	Optimistic
Principle	Local causality constraint violation is strictly avoided; Only safe events are processed.	Allows local causality constraint violation, but can detect and recover to a previous safe state; Processes both safe and unsafe events.
Overheads	Simple simulation executive; May need special mechanism for dynamic LP topology; Lower overheads if good lookahead.	Complex simulation executive requires state saving, fossil collection; Special mechanisms for dynamic memory allocation, I/O, runtime errors.
Parallelism	Limited by worst-case scenario; Requires good lookahead for concurrent execution and scalability.	Model parallelism is fully exploitable.
Deadlock	Deadlock may occur; Need deadlock avoidance strategy, deadlock detection and recovery strategy or synchronous approach.	Deadlock never occurs.
Application development	Relies heavily on lookahead. Potentially complex, fragile code to exploit lookahead.	More robust; Less reliant on lookahead; Greater transparency of the synchronization mechanism.
Memory	Requires less memory.	Requires more memory; State saving overhead; Fossil collection requires efficient and frequent GVT computation; Memory management schemes necessary to prevent memory exhaustion.
Legacy simulators	Straightforward inclusion in federations.	Requires additional mechanisms (e.g., state saving) to support rollback.

2.2 An Overview of the High Level Architecture

2.2.1 Introduction

Starting in March 1995, the High Level Architecture (HLA) [IEEE1516][IEEE1516.1][IEEE1516.2] [IEEE1516.3] was developed by the U.S. Department of Defense (DoD) as a part of the common technical framework (CTF) to facilitate the interoperability and reusability of simulation components. The HLA builds upon the results of the Distributed Interactive Simulation (DIS) standard [DIS94] with the desire to incorporate the advanced time management features that were lacking in the DIS approach. A comprehensive comparison between DIS and HLA can be found in [FUW96]. As the state-of-the-art in the area of distributed simulation, the HLA is being used increasingly in various simulation application areas, including education, training, analysis, engineering, entertainment and games, representing entities at many levels of resolutions.

When talking about the HLA, it is necessary to first introduce the notions of federate and federation: the combined simulation system created from the component simulations is a federation; and each simulation that is combined to form a federation is a federate. The federates can be computer simulations as well as supporting utilities and online simulations with an interface to real systems.

As a standard, HLA is defined by four major elements:

1. The *HLA Rules* define the behavior and capabilities of *federates* and *federations*.

2. The *HLA Object Model Template (OMT)* is used to specify and document the object models that the federates (*Simulation Object Model* or *SOM*) and federations (*Federation Object Model* or *FOM*) have.

3. The *HLA Interface Specification (IFSpec)* defines how the federates communicate with the HLA software which provides the supporting services (*the Runtime Infrastructure* or *RTI*) [RTI02]. The Interface Specification describes six service classes for supporting the HLA federations:

   • *Federation management* – manages creation, control, modification and deletion of a federation execution;
   • *Declaration management* – controls the distribution of object instance attributes and interactions between federates on a *class* basis;
   • *Object management* – manages the creation, modification, deletion of objects, and sending and receiving of interactions;
   • *Ownership management* – manages transfer of instance attribute ownership between federates;
   • *Time management* – controls the advancement of simulated (logical) time;
   • *Data distribution management* – controls the distribution of object instance attributes and interactions between federates on an *instance* basis.

4. The *Federation Development Process (FEDEP)* is a generalized process for building and executing HLA federations. Although part of the HLA standard, it should be noted that the FEDEP is not a requirement. Rather, it is a recommended practice for developing HLA federations. Generally, the top view of the FEDEP model is shown below. The six step process can be implemented depending on the application nature.

   - Define Federation Objectives
   - Develop Federation Conceptual Model
   - Design Federation
   - Develop Federation
   - Integrate and Test Federation
   - Execute Federation and Prepare Results

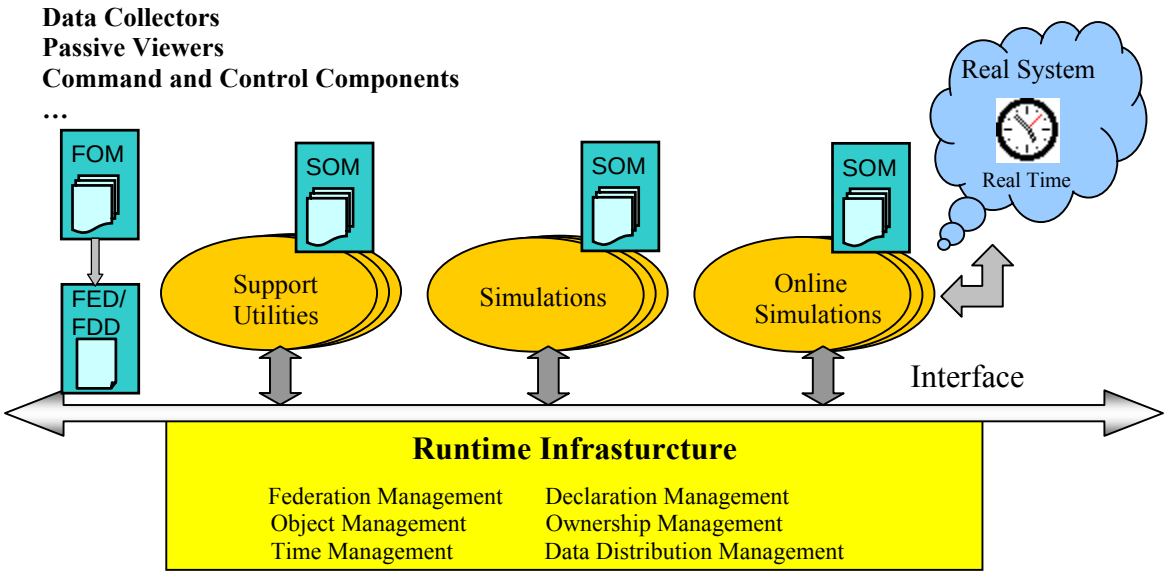


Figure 2.2.1: An overview of an HLA federation

As shown in Figure 2.2.1, an HLA federation is made up of:

- Some federates: each federate is associated with a Simulation Object Model (SOM) which defines the federate's characteristics.
- A Federation Object Model (FOM): the FOM defines the types of and the relationship between the data exchanged between the federates in this federation.
- The Runtime Infrastructure (RTI): The RTI software implements the interface specification and provides services in a manner that is comparable to the way a distributed operating system provides services to applications. With the Federation Execution Data (FED) or FOM Documentation Data (FDD) file, which is a subset of a FOM, the RTI can retrieve all the necessary federation execution details during the creation of a new federation.

The HLA uses an object-oriented view on simulations and simulated entities. All modeled entities are considered as objects with attributes characterizing the objects' states and properties. Changes to the state of an object can be made by updating associated attribute values. Those non-persistent events occurring at certain point in time are called interactions. Interactions have parameters and can be used for modeling interactions and communications between objects. In addition, the HLA defines a bi-directional interface which federates use for communicating with the Runtime Infrastructure (RTI). This interface is based on an ambassador paradigm. A federate communicates with the RTI using its RTI ambassador. Vice versa the RTI communicates with a federate via its federate ambassador. From the federate programmer's point of view these ambassadors are objects and the communications between the participants are performed by calling methods of these objects (see Figure 2.2.2).

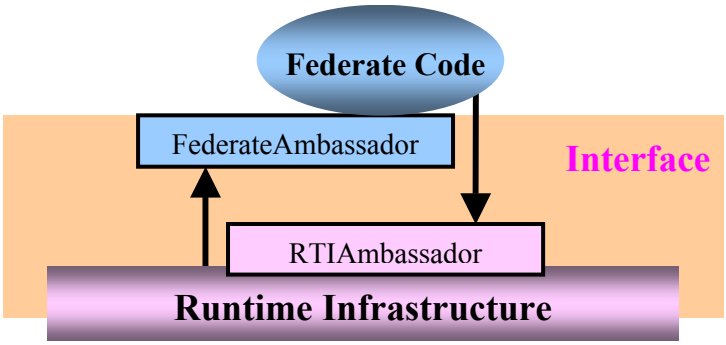


Figure 2.2.2: Interfaces between RTI and federates

2.2.2 Time Management in the HLA

In the HLA, time in the system being modeled is represented in the federation as points along a federation time axis. Each federate, upon joining an execution, is assigned a *logical time*. The logical time is assumed to be zero when the federate first joins the simulation. Time within a federation only advances. Each federate may request to advance only to a time that is greater than or equal to its current logical time. For a federate to advance its logical time, it requests an advance explicitly. The advance will not occur until the RTI issues a grant. Time management is concerned with the mechanisms for controlling the advancement of each federate along the federation time axis. In general, time advances are coordinated with object management services so that information is delivered to the federates in a causally ordered fashion.

2.2.2.1 Message Orders

The HLA provides two general ordering types for messages, named Timestamp Order (TSO) message and Receive Order (RO) message. RO messages are simply placed in a queue when they arrive and are immediately available for delivery to the federate. TSO messages have a timestamp assigned by the sending federate and are placed into a queue within the RTI when they arrive. In conservative synchronization, these messages are not eligible for delivery to the federate until the RTI can guarantee that no more messages with a smaller timestamp will be received by that federate. However, in optimistic synchronization, all the TSO messages can be flushed to the receiving federate which should be able to conduct a rollback procedure when necessary.

2.2.2.2 Time Policies

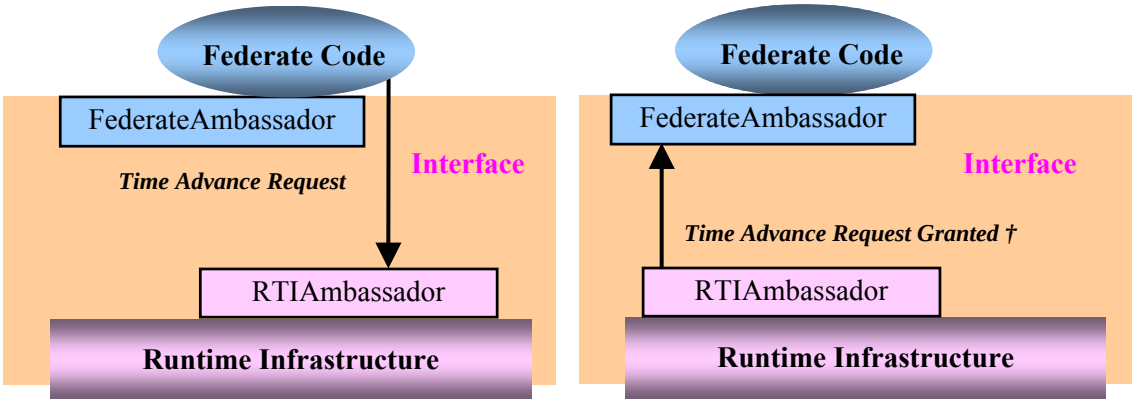
A federation can have federates that use any time policy, time-regulating, time-constrained, both or neither. Each federate adopts the desired time policy associated with its own behavior. Only the time-regulating federate can send TSO messages and only the time-constrained federate can receive TSO

messages. Note that a time-regulating federate need not send TSO messages in timestamp order, but all TSO messages that it sends are received by other federates in timestamp order. Each time-regulating federate provides a lookahead value that establishes a lower bound on the timestamps that can be sent in TSO messages. Each federate in an execution has an associated *lower bound on timestamp* (LBTS) value. The LBTS is calculated by the RTI and represents the smallest timestamp that could ever be received by that federate in a TSO message if that federate is time-constrained. In performing this calculation for a federate, the RTI takes into account the logical time and lookahead of all time-regulating federates in the execution to determine the LBTS value. So the advancement of logical time by a time-regulating federate has an effect on the advancement of time-constrained federates because it acts as its promise not to send any TSO messages with timestamps smaller than some specified time.

2.2.2.3 Transportation Types

The HLA supports both reliable transportation (HLAreliable) and best effort transportation (HLAbestEffort). The reliable transportation guarantees that each message will be received at the specified destination(s), or an exception will be raised for any message that cannot be delivered. Typically the networking software will retransmit the messages if it cannot ensure the message has been successfully received. By comparison, this cannot be guaranteed by best effort transportation. Generally, reliable transportation is preferred for the cases where a lost message can not be tolerated, for example, if it may cause a simulator to enter an inconsistent state. In other cases, such as periodically generated position update messages where occasional lost messages will not bring about significant impact, best effort transportation is preferred since it normally incurs less latency and overhead than reliable services.

2.2.2.4 Time Advance Approaches



(a) (b)

Figure 2.2.3: Time advance in HLA (a) Federate issues the request for time advance to RTI (b) RTI grants the request for time advance to federate

A federate may advance its logical time by explicitly requesting time advancement via the *RTIAmbassdor* and waiting for a grant callback from the *FederateAmbassdor* (shown in Figure 2.2.3). Due to the large diversity of simulations, the RTI offers five services for the federate to advance its logical time (Table 2.2.1)

Table 2.2.1: Time advance services in the HLA/RTI

Time Advance Services	Time Management Approach
Time Advance Request (TAR)	time stepped (conservative)
Time Advance Request Available (TARA)	time stepped (conservative)
Next Event Request (NER)	event driven (conservative)
Next Event Request Available (NERA)	event driven (conservative)
Flush Queue Request (FQR)	optimistic

Whichever time advance service is used, the RTI grants an advance to logical time *T* only when it can guarantee that all TSO messages with timestamps smaller than T (or in some cases smaller than or equal to *T*) have been delivered to the federate (see Table 2.2.2). This guarantee enables the federate to simulate the behaviors of the entities it represents up to logical time *T* without concern for receiving new events with timestamp smaller than *T*.

Table 2.2.2: Time advance service descriptions (reproduced from [DSS00])

Time advance services	Constraint on advance to t_1	Messages delivered before grant to t_2	Constraint on grant to t_2	
TAR (non-zero lookahead)	Cannot send $t_s < t_1$ + lookahead	All queued RO messages. All TSO messages with $t_s \leq t_2$.	Cannot send $t_s < t_2$ + lookahead	$t_2 = t_1$
TAR (zero lookahead)	Cannot send $t_s \leq t_1$	All queued RO messages. All TSO messages with $t_s \leq t_2$.	Cannot send $t_s \leq t_2$	$t_2 = t_1$
TARA	Cannot send $t_s < t_1$ + lookahead	All queued RO messages. All TSO messages with $t_s < t_2$. All queued TSO messages with $t_s = t_2$.	Cannot send $t_s < t_2$ + lookahead	$t_2 = t_1$
NER (non-zero lookahead)	Cannot send $t_s < t_1$ + lookahead	All queued RO messages. Smallest TSO message that will ever be received that has a $t_s \leq t_1$ and all other TSO messages with the same t_s.	Cannot send $t_s < t_2$ + lookahead	$t_2 \leq t_1$
NER (zero lookahead)	Cannot send $t_s \leq t_1$	All queued RO messages. Smallest TSO message that will ever be received that has a $t_s \leq t_1$ and all other TSO messages with the same t_s.	Cannot send $t_s \leq t_2$	$t_2 \leq t_1$
NERA	Cannot send $t_s < t_1$ + lookahead	All queued RO messages. Smallest TSO message that will ever be received that has a $t_s \leq t_1$ and all other TSO messages with the same t_s.	Cannot send $t_s < t_2$ + lookahead	$t_2 \leq t_1$
FQR	Cannot send $t_s < t_1$ + lookahead	All queued RO messages. All queued TSO messages.	Cannot send $t_s < t_2$ + lookahead	$t_2 \leq t_1$

An attractive property of the RTI is that it supports a transparent time advance approach, which means each federate need not understand the local time management structure of other federates. This is a key to achieve interoperability among federates using different local time management mechanisms.

2.3 Existing Approaches for Integrating CSPs with the HLA

Commercial Off-The-Shelf (COTS) simulation packages (CSPs) are used for many purposes, including analysis, training, prototyping, etc. Examples of CSPs include: ProModel [HAP03], Arena [BAS03], AutoMod [ROH03], Simul8 [SIMUL8], Extend [KRA03] and Witness [WITNESS].

For some applications a standalone system is sufficient [COX98]. With the benefits of supporting reuse and interoperability, however, distributed simulation has become increasingly important. Other reasons, such as scalability, information hiding and group working also motivate the use of distributed simulation systems.

2.3.1 Requirements

Some requirements are imposed for the CSPs to support a distributed simulation. The requirements mainly come from the data exchange and representation, time synchronization and the HLA programming paradigm.

- Support for Data Exchange and Representation

The HLA uses an object-oriented view on simulations and simulated entities. All modeled entities are considered as objects with attributes or interactions with parameters. However, many simulation practitioners outside of the defense area tend to use a CSP rather than object-oriented, or at least object-based, software. Additionally, most of the formats used by these packages are incompatible (a model saved by one cannot be opened by another) [MFT01]. So it is important to consider how the Object Model Template (OMT) can be used to represent model information that must be shared between models running in distributed CSPs [TBF02][TAY02]. The High Level Architecture Commercial Off-The-Shelf Simulation Package Integration Forum (HLA-CSPIF) [HLA-CSPIF] was established in 2002, with the aim to create a standardized approach to distributed simulation using HLA. One of the objectives of the forum is to develop a standard data exchange representation based on the proposed standard interoperability reference models (IRMs). In 2004, the HLA-CSPIF was awarded Product Development Group Status and was approved as the CSPI-PDG (Commercial Off-

The-Shelf Simulation Package Interoperability Product Development Group) by the Simulation Interoperability Standards Organization (SISO).

- Support for Time Synchronization

Being part of a distributed simulation, each simulation component must synchronize itself with other components. The HLA supports simulation systems which are based on different synchronization approaches. It offers the possibility to combine simulation systems each of which use a different approach. To make a simulation system use one of the approaches, it must fulfill certain requirements listed in Table 2.3.1.

Table 2.3.1: General synchronization approaches and their requirements on the simulation systems (reproduced from [STR01])

Synchronization Scheme	Some Imposed Requirements
Conservative Synchronization	The system must have a built-in function to decide the timestamp of the next event as well as the capability of pausing the simulation to wait for other federates.
Optimistic Synchronization	The system must be able to save the state of the simulation periodically to allow possible roll back to a past state. It must be capable of computing the value of LBTS. Moreover, it must has the capability to detect local causality violation.
Time-stepped Synchronization	The model must be capable of taking inaccuracies introduced by using fixed time steps into account.
Real-time Synchronization	The system must have the capability to pace its simulation time according to wall clock time (does not work for slower-than-real-time simulations). This approach is more suitable for simulations which do not require strict causality.

- Support for HLA communication paradigm

As discussed in section 2.2, the HLA comes with its own programming paradigm with which the simulation system must fit. During the normal execution of a federate, there is a need for communications in two directions: from the federate to the RTI and from the RTI to the federate. This means the simulation model should be able to call the functions of the *RTIAmbassador* and receive callbacks from its own *FederateAmbassador*. Some CSPs provide open source code (e.g., Simplex3 [SCH00]) or an open interface (e.g., SLX [HEN97]) which can be used for constructing calls to the *RTIAmbassador* methods. But most of the CSPs do not support the capability of defining the user's callback functions. Therefore, alternative ways must be adopted to directly access the memory address of objects and interactions defined in the user's model. Because in HLA the objects and interactions are registered and initiated at runtime, the memory address of each object / interaction cannot be determined in advance. Some CSPs (e.g., SLX) offer the option of querying its

symbol table through its own callback function to allow external interfaces to determine the structure of each class at runtime. For other simulation tools which do not offer this capability, additional information should be passed to the HLA interface.

2.3.2 Implementation Approaches

In [STR01], the HLA interfaces for simulation systems are classified with respect to two distinct perspectives: the programmer's perspective (the person developing the HLA interface and performing the low level programming) and the user's perspective (the person developing the HLA based models).

The Programmer's Point of View

- Re-implementation of the tool: the source code of a simulation system is available. The most concern for this approach is the compatibility of the programming languages used by the CSP and the RTI library. For example, if a CSP is written in C, it needs to carefully implement the linkage with the C++ or Java version of the RTI library.
- Extension of intermediate code: the tools translate model descriptions written in a tool-dependent modeling language in another programming language. This makes it possible to modify the intermediate code to integrate with the HLA.
- Usage of an external programming interface: the tools offer an open and extensible architecture. The programming of the interface to the RTI can use the DLL (Dynamic Library Link) interface, OLE (Object Linking and Embedding), DCOM (Distributed Component Object Model), ActiveX, etc.
- Coupling via a gateway program: the tools cannot access the HLA API by any of the prior methods. The gateway program can communicate with the CSP via appropriate means (e.g., file, pipe, port and network) depending on the capability of the CSP.

The User's Point of View

- The explicit approach: a library interface of the simulation tool is used to provide HLA functionality. The modeler needs to enhance the model with HLA functionality by calling functions in the library that correspond with the methods defined in the HLA interface specification.
- The implicit approach: all HLA functionality is hidden from the model developer since the CSP and its underlying software handle all the HLA synchronization and communication. Such a below-the-surface approach can easily be applied if the source code of the simulation system is available. For example, an implicit approach could be used for Simplex3 because its source code is available.

2.3.3 Survey of Current HLA Interfaces for CSPs

To meet the demand for interoperability and reusability of simulation models using CSPs, a number of HLA interfaces for CSP have been developed. These interfaces make the CSPs HLA compatible, allowing them to be integrated into a distributed simulation environment. Table 2.3.2 briefly describes some typical HLA interfaces and compares their implementation approaches as well as their time management mechanisms (part of the table is compiled from [STR01]). The diversity of the CSP properties and extensibility leads to various implementation approaches, which motivates us to propose a generic architecture for the integration of CSPs and the HLA.

Table 2.3.2: Survey of current HLA interfaces for CSPs

Tools	Features available for coupling with HLA	Ways for extending with HLA		Time Management			Status
		Programmer's Point of View	User's Point of View	Features	Synchronization Approach	Lookahead	
SLX [HEN95] [HEN96] [HEN97] [HEN99] [HEN00]	o It has a library interface which allows calling functions in any DLL from SLX. o It offers the feature of writing DLL header files, which disclose the internal structure of SLX objects. o It has some query functions used to determine the composition and the memory layout of SLX data structures, followed by a function which notifies SLX about the occurrence of an external event.	External Programming Interface (Provides a DLL library file and a header file to the user)	Explicit (The user need explicitly call the SLX-HLA interface functions.)	Built-in function: *next-imminent-time*()	Conservative	Specified by the user when calling RTI-Init()	Released
Simplex3 [SCH00]	Source code written in C is available	Re-implementation	Implicit (The user only need provide a mapping file)	Since source code is available, internal event lists could be accessed to determine the timestamp of the next event.	Conservative	Zero lookahead	Experimental
Pro Model [RPH99] [HAP00] [HAF01]	It offers a function called XSUB(), which could be used to call external functions inside a Windows DLL	External Programming Interface	Explicit	It does not provide a built-in function for accessing the event timestamp of future events.	Time-stepped (TAR)	Specified by the user when calling RTI-Init()	Experimental
MODSIMIII [JOH99]	o It was equipped with an HLA interface by its products company. The raw HLA API is packaged into MODSIM objects that support all HLA areas. The HLA interface provides automatic handling of message exchange, exceptions and callbacks. o It provides a universal data value representation so that data can be transported via the network between different computer platforms with different endian types. [BLE]	Re-implementation	Explicit	Not mentioned	Not mentioned	Not mentioned	Released

AutoMod [STA01] [ROM02]	Included in the AutoMod software suite is the Model Communications Module (MCM), which allows an executing simulation to open socket connections and to send and receive messages via TCP/IP network protocol. [HIF02]	External Programming Interface (It uses a wrapper library like SLX and Pro Model.)	Explicit	It does not provide a built-in function for accessing the event timestamp of future events.	Time-stepped (TAR)	Not mentioned	Experimental
DEVS [ZBC99] [STR01]	The normal communication between DEVS models takes place via input and output ports.	Re-implementation	Close to Implicit (Certain tasks done in a programming language remain up to the user.)	Each federate starts by using its time advance function to compute its time-of-next-event, t_n, and sending it in the form of a *nextEventRequest* with cutoff parameter = t_n.	Conservative	Not mentioned	Released
MATLAB [PLD00]	It provides MEX interface to allow the user to write C and FORTRAN programs that interact with MATLAB. The MATLAB API includes facilities for calling routines from MATLAB (dynamic linking) and for calling MATLAB as a computational engine (callback mechanism).	External Programming Interface	Explicit	Not mentioned	Not mentioned	Not mentioned	Experimental
Manufacturing Adapter [HIF02]	o It provides access to RTI functionality through a socket interface. It acts as a gateway program between simulators and RTI. o It transforms XML format messages between manufacturing simulator and HLA/RTI. o It uses a Relational Data Definition (RDD) file which defines exchanged message contents between the simulators. o It uses the socket protocol for communication with manufacturing system simulator.	Gateway Program	Explicit	A function to send a simulation clock from the manufacturing system simulator to HLA/RTI using the TAR method and NER method.	Conservative (NER) Time-stepped (TAR)	Specified by the user in RDD file using XML	Released

AnyLogic [ALOGIC] [BKK02]	To make AnyLogic HLA-compatible, a special module, HLA Support Module (HSM) is developed by the products company to route messages and synchronize the local system clock to federation global time. The AnyLogic kernel interacts with RTI though the HSM. A special interface named StepHook is used as an interface between the HSM and AnyLogic Engine. The StepHook interface allows the developer to put a hook on the engine performing model time steps.	Re-implementation	Explicit	AnyLogic provides a set of predefined classes, inherited from HLA Support Module class library, for the user to build HLA-compatible models.	Conservative	Not mentioned	Released

2.4 CSPI-PDG Interoperability Reference Models

As discussed previously, the HLA is chosen as the communication standard for the interoperability of the CSPs. However, there are still some roadblocks that need to be tackled to achieve the integration of the CSP with the HLA. One problem is the difficulty to match model information represented in CSPs to the object/interaction concept in the HLA standard. Another problem is that terminology between different packages differs as there is no internationally recognized naming convention. To address the two problems, some general Interoperability Reference Models (IRMs) and an entity transfer specification using OMT in the HLA are proposed.

2.4.1 Interoperability Reference Models

The CSPI-PDG Interoperability Reference Models (IRMs) are one of the set of products proposed by the CSPI-PDG [TTL05]. The aim of the IRMs is to categorize the integration problems into different requirements, thereby providing an easy way to create solutions for each specific integration problem. Although these models are conceived for the industrial domain, similar approaches could be derived from them for other application domains.

The purpose of the IRMs is embodied in two points: (1) to capture the essence of real modeling problems, and (2) to facilitate communication between vendors, modelers and technologists on distributed simulation issues. Figure 2.4.1 shows the initial model that might be built in a CSP. All the figures of the IRMs in this section are reproduced from [TAY03].

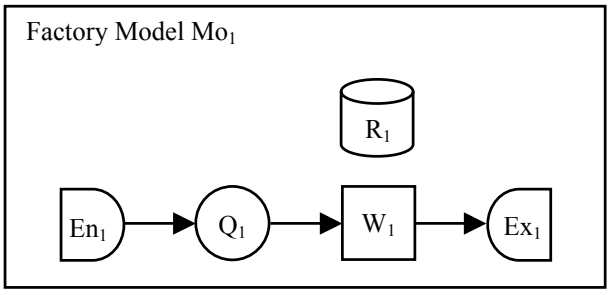


Figure 2.4.1: Initial reference model

This model is composed of an entry point (arrival source) En_1, a queue Q_1, a workstation W_1, a resource R_1, and an exit point (exit sink) Ex_1. The entities passing through this factory are also known as parts. Entities have some attributes, for example, Type, Entry time, and Exit time. Entities (parts, packets, flow items, and so on) arrive in the factory at the entry point En_1 according to an arrival time distribution and are placed in the queue Q_1. Entities wait in Q_1 until the workstation W_1 is free and

not undergoing repair. If W_1 is free, an entity is loaded and processed according to a processing time/part type distribution. When the processing is finished, the entity exits the factory via the exit point Ex_1 and, if there are entities waiting in Q_1 and the workstation does not break down, another entity will be processed. If the workstation breaks down a repairman must be available and will repair it according to a workstation repair distribution. The repairman is modeled by resource R_1. Various details such as arrival time distribution, processing time distribution, workstation breakdown distribution, are described by using menus associated with each model element. All distributions can be either deterministic or stochastic.

In a HLA distributed simulation, each simulation component is built using a specific CSP as a federate that interacts with other federates. There are currently six types of reference model and each type represents a modeling requirement that presents a particular challenge to distributed simulation. Here, the federation consists of the two federates (Fd_1 and Fd_2) that run on different computers connected by a network and distributed simulation software (an RTI). The information exchanged between the federates is the various entities generated by the first factory and consumed by the second one.

Figure 2.4.2 describes the first type, named asynchronous entity passing or basic interoperability reference model. Here 'asynchronous' means there is no immediate or direct feedback when an entity is passed. The exit point Ex_1 in Mo_1 and the entry point En_2 in Mo_2 are connected by a direct link agreed with the stakeholders of both factories. The combined models represent the combined factory as entities finishing the process in W1 are transferred directly to queue Q_2 to wait for the process in W_2. In this thesis, the entry point (En_2) designed to receive entities from external models is called an 'external entry point' and given the abbreviation 'EEP'. On the sending side (Mo_1), the EEP (En_2) in the receiving model (Mo_2) is referred to as the remote EEP, and on the receiving side (Mo_2) the EEP (En_2) is referred to as the local EEP.

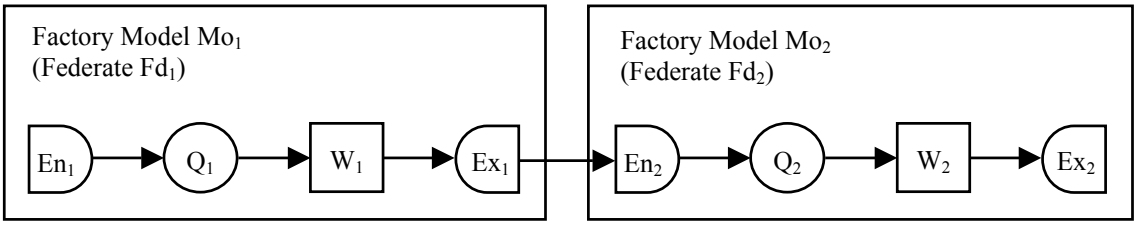


Figure 2.4.2: IRM Type I: asynchronous entity passing

Figure 2.4.3 describes the second type, named synchronous entity passing or bounded buffer interoperability reference model. In this type, the queue Q_2 is now bounded: when Q_2 becomes full, W_1 cannot send any more entities to Q_2. It indicates the requirement that Mo1 containing the sending workstation W_1 must, when the processing of an entity is complete, check to determine that there is

space in Q_2. If there is space available then the entity may be transferred. Otherwise M_1 must ensure that W_1 is blocked until space becomes available. A similar case is that En_2 directly links with a workstation with limited capacity, which may also block W_1 in Mo_1.

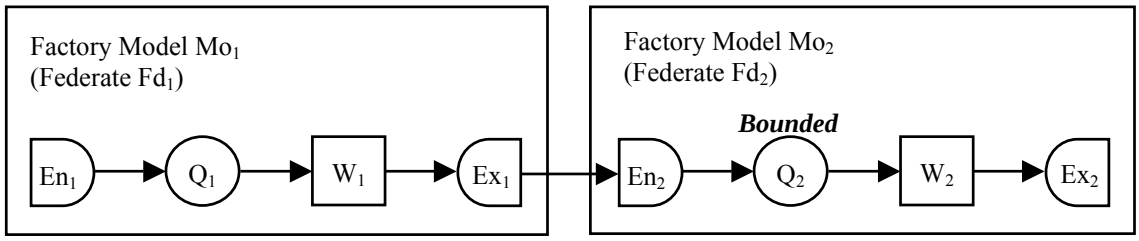


Figure 2.4.3: IRM Type II: synchronous entity passing

Figure 2.4.4 describes the third type, named shared resources interoperability reference model. In this type, resource R is shared between the two models as a global or shared resource. If W_1 breaks down, it will be repaired by R according to a workstation repair distribution. If W_2 also breaks down when R is utilized by W_1, it must wait until R is released by W_1. The state information of R should be visible to Mo_1 and Mo_2 for enquiry.

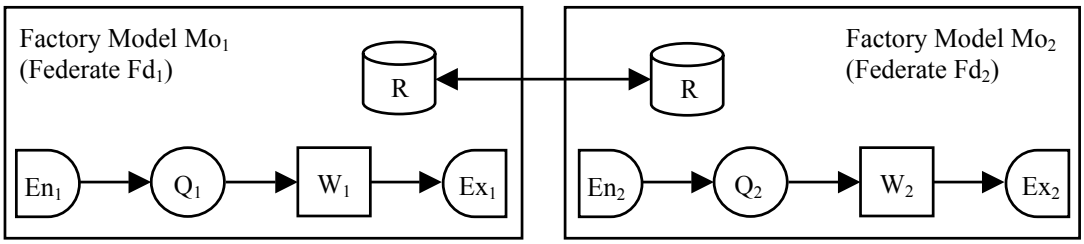


Figure 2.4.4: IRM Type III: shared resources

Figure 2.4.5 describes the fourth type, named shared events interoperability reference model. The event *E* is shared in Mo_1 and Mo_2. This is intended to be a shared "signal" between the two models. For example, when model Mo_2 reaches a given threshold value (a quantity of production) it should be able to signal this fact to all models that have an interest in this information.

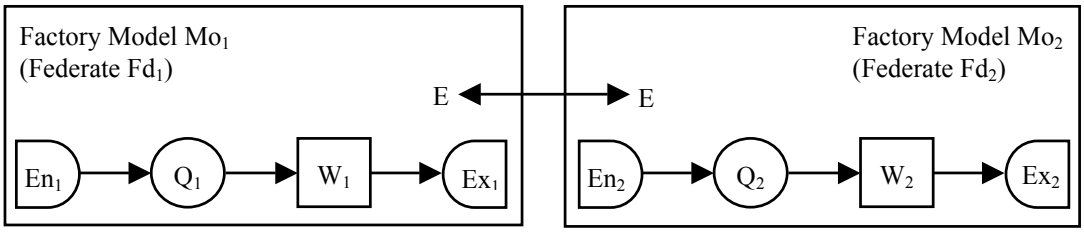


Figure 2.4.5: IRM Type IV: shared events

Figure 2.4.6 describes the fifth type, named shared data structures interoperability reference model. The data structure D that is shared between Mo_1 and Mo_2. This can be used to represent any data that

needs to be shared between the models. Any update to one copy of D has to be synchronized with all copies of D.

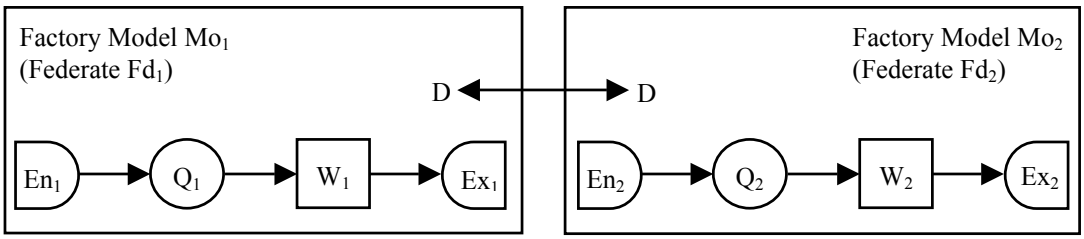


Figure 2.4.6: IRM Type V: shared data structures

Figure 2.4.7 describes the sixth type, named shared conveyors interoperability reference model. Conveyor C is shared between the two models so that the exact position of each entity is shown correctly. The combined models represent the combined factory where parts finishing machining in W_1 are transferred directly on a shared conveyor C to be scheduled in Q_2.

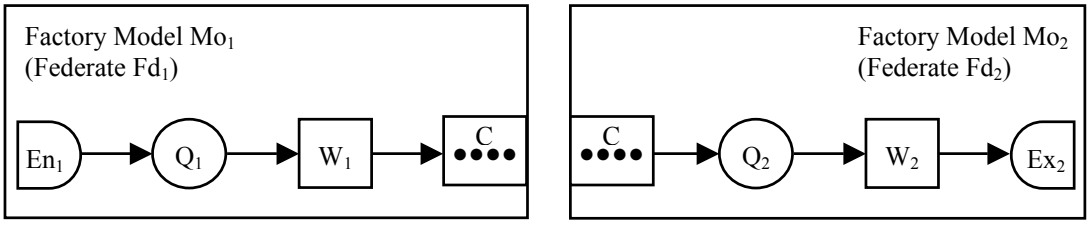


Figure 2.4.7: IRM Type VI: shared conveyors

2.4.2 Entity Transfer Specification Representation in the OMT

The Entity Transfer Specification (ETS) in this thesis deals with the representation of entities in Type I and Type II IRMs. The difference between the IRMs is that Type II requires additional synchronization to deal with the synchronous entity passing problem.

An *entity* is an element of the system being simulated that can be individually identified and processed, for example, a job waiting to be processed on some workstation [PID04]. In CSP distributed simulation, the entities define the information flow (exchange) between federates in a federation. An entity may consist of a name, a set of attributes with data-types, semiotic values (documentation) and a timestamp. The information exchanged between federates must be represented according to the HLA requirements.

In the HLA, all federations and individual federates are required to be represented by an object model that identifies the data to be exchanged during the execution of a federation/federate [TAY02]. The Object Model Template (OMT) specification is used to define the format and syntax (but not the content) of the HLA object models. The object model used to describe a federation is called the Federation Object Model (FOM), and the object model used to describe a federate is called the Simulation Object Model (SOM).

In the HLA, object models are represented as object classes (with particular members, each referred to as an instance) or interaction classes. The shared information between federates is transferred indirectly via HLA services by updating attributes values of objects or by sending interactions with parameters. In the OMT, the HLA object models are composed of a group of interrelated components specifying information about classes of objects and their attributes and interactions and their parameters. Some crucial components [IEEEP1516.2] are listed as follows:

- *Object class structure table*: to record the namespace of all federate or federation object classes and to describe their class-subclass relationships.
- *Interaction class structure table*: to record the namespace of all federate or federation interaction classes and to describe their class-subclass relationships.
- *Attribute table*: to specify features of object attributes in a federate or federation.
- *Parameter table*: to specify features of interaction parameters in a federate or federation.
- *Datatype tables*: to specify details of data representation in the object model.

In [TYT04] three different approaches to realize entity transfer are examined. The first approach involves using object updates to realize entity transfer between federates. The second approach achieves entity transfer by sending interactions. In contrast to the first approach which assumes static object attributes, the third approach assumes dynamic object attributes, which are attributes of entity objects whose ownership can be transferred to another federate during the object's life cycle. Benchmark simulation results using the DMSO (Defense Modeling and Simulation Office) RTI [DMSO00][RTI02] show that entity transfer by sending interactions is more efficient than object updates when the number of attributes of the entity is large. In the generic architecture, the entity transfer is represented by sending interaction. More details will be discussed in Chapter 3.

2.5 Summary

This chapter first reviews research in the areas of parallel and distributed simulation. It focuses on time management, including the concepts of different types of time, lookahead, and the two main synchronization approaches, referred to as conservative synchronization and optimistic synchronization. Many algorithms have been proposed for each synchronization approach to find an optimal protocol. However, the selection of the conservative or optimistic synchronization approaches is simulation scenario dependent. The work described in this thesis considers both of the two approaches and aims to introduce them into the integration of the CSPs. The High Level Architecture (HLA) is also introduced as the communication standard to support the integration of the CSPs. In addition to the general framework and major elements, the time management services provided by the HLA are explained. Some work has been done to successfully enable the CSPs to be HLA compatible. This is summarized in terms of their implementation approaches and time management, which offer the basis for the proposed generic architecture for integrating the CSP with the HLA. Moreover, the six types of CSPI-PDG IRMs as well as an entity transfer specification using OMT in the HLA are described. Solutions for the problems of Type I and Type II IRMs will be proposed in Chapter 3.

CHAPTER

3

A GENERIC ARCHITECTURE FOR INTEGRATING CSPS WITH THE HLA

3.1 Introduction

With the development of CSPs and the advent of the HLA, it is possible to interconnect models developed using specific CSPs best suited to the application area. However, different CSPs have diverse characteristics and different degrees of extensibility, which makes the integration of the CSPs difficult. Besides, the complexity of the HLA standard itself and the lack of expertise in distributed simulation (e.g. time synchronization algorithms) also become a roadblock to widespread adoption of the HLA in linking CSPs. In recent years, researchers have investigated and proposed some CSP-HLA interfaces to integrate some specific CSPs with the HLA. In Section 2.3.3, the major features of these CSP-HLA interfaces are summarized. These interfaces make the CSPs HLA compatible, allowing them to be integrated into a distributed simulation environment. The interfaces can be classified as either explicit or implicit from the modeler's point of view [STR01]. In the explicit approach, the modeler needs to enhance the model with HLA functionality by calling functions that correspond with the methods defined in the HLA interface specification. In the implicit approach, all HLA functionality is hidden from the modeler since the CSP and its underlying software handle all the HLA synchronization and communication. Obviously, the implicit approach makes it easier for the modeler to link simulation models together. Some drawbacks, however, exist in current CSP-HLA interfaces:

- Most current interfaces adopt the explicit approach from the modeler's point of view. These include the interfaces for SLX [SSK98], Pro Model [STR01], MODSIM III [JOH99], MATLAB [PLD00], RTI-Adaptor [STR01] and Manufacturing Adapter [HIF02]. Since the

implicit approach usually requires the availability of the source code of the CSPs to integrate HLA functionality, it is provided by few CSPs.

- Another restriction is that each interface is designed for a particular CSP, since currently existing CSPs are heterogeneous in terms of their properties and extensibility. Different CSPs have different degrees of capabilities for their external interfaces. The CSP-HLA interface designed for one CSP may not be suitable for others. It is troublesome to develop a separate CSP-HLA interface for each CSP.

- The third drawback comes from the fact that the interfaces are usually based on the conservative approach since it is easy to implement. However, the optimistic approach can achieve better performance in situations where lookahead is difficult to exploit, as the need for lookahead is the main constraint for the conservative approach.

To overcome these drawbacks, a generic interface for integrating various CSPs with the HLA is required. Even though a totally implicit approach is preferred, it is difficult to acquire source code for the CSPs. However, if a generic interface which is easy to understand and link is formalized for the CSP, the CSP can provide the necessary integration task on behalf of the simulation modelers. In an ideal situation, the integration can be realized without too much involvement of both the CSP and the modeler. That is why a middleware approach is introduced in the generic architecture. This middleware is mainly responsible for time management and data exchange between different models in the distributed simulation with the objective to reduce the involvement of the CSP and the model to as little as possible. A problem here is that the generic middleware cannot have full knowledgement of the structure and the dynamic status of a specific model as well as the internal simulation engine of a certain CSP. For example, the next event time should be forwarded to the RTI to advance simulation time, which should be provided by the CSP. A compromise is to keep the middleware generic for heterogeneous CSPs while requiring the CSPs to pass some necessary integration information. Furthermore, the interface should support different synchronization approaches which can be applied to different types of simulation scenarios. Meanwhile the complicated details of time management can be transparent to the CSPs as much as possible. In this way, the simulation modeler can concentrate on the particulars of their simulation models, making the application more robust. They will not need to expend a great amount of effort to make their models part of a distributed simulation.

3.2 The Architecture

In section 2.3.1, the requirements for integration of a CSP with the HLA are analyzed from two aspects: requirements resulting from being part of a distributed simulation and requirements resulting from a certain programming paradigm. Being part of a distributed simulation indicates that the CSP should provide an interface to connect to other CSPs or simulation programs. This architecture also requires a common standard for data representation and exchange as well as a synchronization mechanism between different simulation components. For the programming paradigm, on the other hand, special consideration is needed because of the ambassador paradigm of the HLA. The CSP should be able to communicate with the HLA runtime infrastructure (RTI) through the RTIAmbassador and implement the corresponding FederateAmbassador.

3.2.1 Overview of the Architecture

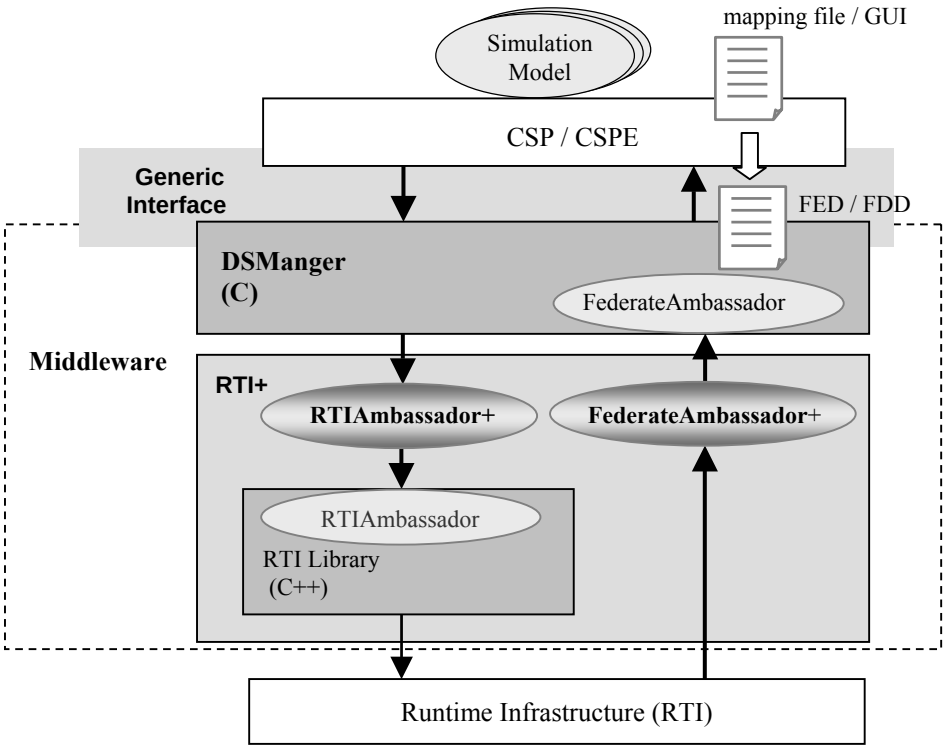


Figure 3.2.1: A generic architecture for integrating a CSP with the HLA

Based on the requirements analysis [STR01], a generic architecture is proposed with the incorporation of middleware as shown in Figure 3.2.1. The middleware is made up of two layers, the DSManager library and the extended RTI. The upper layer, the DSManager, provides a generic interface consisting of a set of functions to be invoked by the CSP or CSP Emulator (see Chapter 4) when necessary. The C++ / Java based RTI is wrapped by “normal” C functions that can easily be

integrated with most of the current CSPs written in C, C++, Java or VB. Another important feature of the DSManager is to try to hide the HLA concept from both the CSP and the modeller. It is difficult to match model information represented in the CSPs to the object/interaction concept in the HLA standard. In addition, the terminology between different CSPs differs as there is no internationally recognized naming convention. The interface adopts a generic approach based on the concept of entity transfer, and will be proposed as a standard by the CSPI-PDG in the future.

In the architecture, the DSManager also interacts with the lower layer of the middleware, the extended RTI. In this extended RTI, known as RTI+, appropriate synchronization algorithms are designed in order to improve the simulation performance and relieve the user from the burden of time management. Examples are a shared state manager [GLW03][LGW03] (see Chapter 6) for conservative synchronization and a rollback controller (see Chapter 5) for optimistic synchronization. The RTI+ library is composed of the RTIAmbassador+, the FederateAmbassador+ and the original RTI library. It provides all the services in the RTIAmbassador as well as some new or modified/extended services and filters the callbacks through the FederateAmbassador+ before passing them to the FederateAmbassador. It is important that all necessary implementation in the RTI+ is transparent to the DSManager, CSP and the modeler. The RTI+ library is linked with the DSManager instead of the usual RTI library.

The implementation of the generic architecture is based on the cooperation between the model, the CSP, the DSManager and the RTI+. Although the DSManager and the RTI+ are responsible for interacting with other simulation components within the distributed simulation, the model and the CSP have to fulfill some requirements to be part of the distributed simulation. In the next section, the general requirements for all types of CSPI-PDG IRMs are discussed.

3.2.2 Requirements for the Model

One feature of the generic architecture is to use an implicit approach from the modeler's point of view. Here, "implicit" means that all the HLA functionality is hidden from the simulation model developer. To be part of a distributed simulation, the model only needs to supply some necessary information for data exchange with other simulation components. A modeler can build a standalone simulation model without the DSManager or a sub-model that is a constituent of a distributed simulation. If the HLA functionality is required, the simulation model must provide some necessary information to the CSP in the initialization phase. The information mainly includes the following parts:

(1) Mapping information

The mapping information can be provided as a file or via a user friendly GUI provided by the CSPs [RYT03]. It specifies the name of the federate and the federation which it will join. It also declares what kind of information the federate wants to exchange with other federates. The mapping information is transferred from the CSP to the DSManager and is automatically converted to the corresponding FED (Federation Execution Data) [DOD98] or FDD (FOM Document Data) [IEEE1516] file, which is required as an input to the RTI. For data published by the simulation model, appropriate object registrations, updates and interactions are also automatically generated at runtime. This is realized by triggering the relevant RTI calls in the DSManager when the value of a variable in the simulation model changes. The CSP is in charge of monitoring the modification of the variables during the simulation run. The mapping information is also used by the DSManager when receiving data and delivering it to the appropriate simulation model variables. It must be careful to find the right mapping between object/interaction and data inside the simulation model. For Type I and II IRMs desribed in this thesis, interaction classes are used to map the entities transferred between the simulation models. In future work, however, object classes may also be used to represent some shared variables.

(2) Time policy

In the HLA, a given federate can be *time-regulating*, *time-constrained*, *time-regulating and time-constrained*, or *neither time-regulating nor time-constrained* (the default time policy). The specified time policy should be based on whether the model federate subordinates itself to the federation time advancement or acts as regulating on other federates. Since the time policy is application dependent, it should be specified by the user according to the properties of the simulation model.

(3) Synchronization approach

In a standalone simulation the simulation clock resides within the CSP itself. In a distributed simulation, the synchronization of simulation clocks is usually necessary since strict causality must be obeyed. In general, synchronization algorithms can be divided into either conservative or optimistic approaches. Compared to the conservative approach, the optimistic approach is generally more complex because of the need for state saving and rollback. The best approach should be chosen based on the specific simulation scenario which is unknown by the DSManager and the RTI+. This information should be provided by the modeler. However, the cumbersome management of these mechanisms will be implemented in the RTI+ transparently.

In summary, the generic architecture is implemented using an almost totally implicit approach from the modeler's point of view. Although the modelers are still required to provide some information, they need not worry about the efforts of learning the HLA. The HLA functionality will be fulfilled through the coordination between the CSP and the DSManager, which is hidden from the modelers.

3.2.3 Requirements for the CSP

As shown in Figure 3.2.1, the CSP is responsible for communicating with the DSManager on behalf of the simulation model. To keep the modelers' involvement as small as possible, some requirements should be imposed on CSPs as follows:

(1) External features for the model

The CSP should enable the modeler to design interoperating models through a GUI or a specified file. The user interface should be friendly enough for the modelers to develop their model components in a "plug & play" way without requiring much knowledge about the HLA and the interoperability issues. But that does not mean the modelers are totally free from the task of integration. The model component needs to tell the CSP what information it wants to exchange with other model components as well as other interoperability information (e.g., synchronization approach, lookahead value). Moreover, the CSP should be able to interpret this information and forward it to the DSManager to join a distributed simulation.

(2) Data Conversion

The CSP should take care of the conversion between CSP specific data types and data types as defined in the FOM. The same data types in heterogeneous CSPs may have different sizes, and different hardware platforms may have different endian-types [BLE] and word alignment. So these CSPs should agree on the representation in the corresponding FOM. For simple base types like integer or double, the conversion is relatively simple. More consideration should be taken for complex data structures, since these data structures are transmitted as a sequence of bytes. For the proper decoding process, the corresponding data types of the data structures and their byte sizes and positions must be known.

(3) Communication with the DSManager

In the HLA paradigm, the federate needs to invoke the services of the RTI ambassador and later receive callbacks through its own federate ambassador. This so-called callback mechanism is used for receiving all kinds of data from the RTI, including the interactions, general status information and

information regarding the time management. To make the communication as simple as possible, the two-part interface is wrapped by the DSManager and only a one-direction C interface is provided to the CSP. The CSP passes all the necessary information supplied by the model to the DSManager and gets the returned results from the DSManager. For example, when the model needs to advance time, the CSP will invoke the corresponding function with the required simulation time and wait until the granted time is returned by the DSManager.

In other words, to support the HLA based distributed simulation, the CSP requires small modifications to both the external features (interaction with the modeler) and the internal simulation system (communication with the underlying DSManager). With the use of the DSManager, the modification will not put a large burden on the CSP. In Chapters 4 and 7, there will be further discussions of the integration of the HLA with the CSPE and with a real CSP, IBM's WBI Modeler.

3.3 DSManager for CSPI-PDG Type I IRM

Currently, the DSManager supports the CSPI-PDG Type I IRM and Type II IRM. The DSManager for the Type I IRM focuses on the general interoperability problem, including entity representation, information exchange, and time management. The DSManager for the Type II IRM has some additional features to solve the problems introduced by synchronous entity passing.

3.3.1 The Interface Provided by the DSManager

The DSManager provides an interface consisting of a set of functions to be invoked by the CSP when a distributed simulation is created. Through the interface, the DSManager invokes necessary calls to the RTIAmbassador+ on behalf of the CSP and transfers the information received from the FederateAmbassador+ to the CSP. Figure 3.3.1 shows the basic communication protocol between the CSP, DSManager and RTI+ for CSPI-PDG Type I IRM.

In the initialization phase, the CSP registers its model as part of a distributed simulation via *registerDS*. Receiving such information, the DSManager invokes the corresponding *createFederationExecution* and *joinFederationExecution* in the RTIAmbassador+. One of the component models must be assigned as a controller in a distributed simulation. It is responsible for synchronizing the start of the distributed simulation after all the component models have joined the federation. For the controller model in the distributed simulation, *registerController* is also invoked. The controller model will count the number of federates that have joined the federation. When all

federates have joined, it will invoke some RTI+ services to achieve synchronization of the starting point.

Since the component model needs to interact with other component models, it should have some input and/or output information. In the generic interface, such information is represented using the concept of entity. The functions *registerInEntity* and *registerOutEntity* are provided for the CSP to declare an exchanged entity. For each exchanged entity, the modeler should give the entity name and the name of the external model with which the entity is exchanged. This information is passed to the RTIAmbassador+ by calling *subscribeInteractionClass* and *publishInteractionClass*. As discussed in Section 2.4.2, an interaction class rather than an object class is used to transfer an entity in the interface.

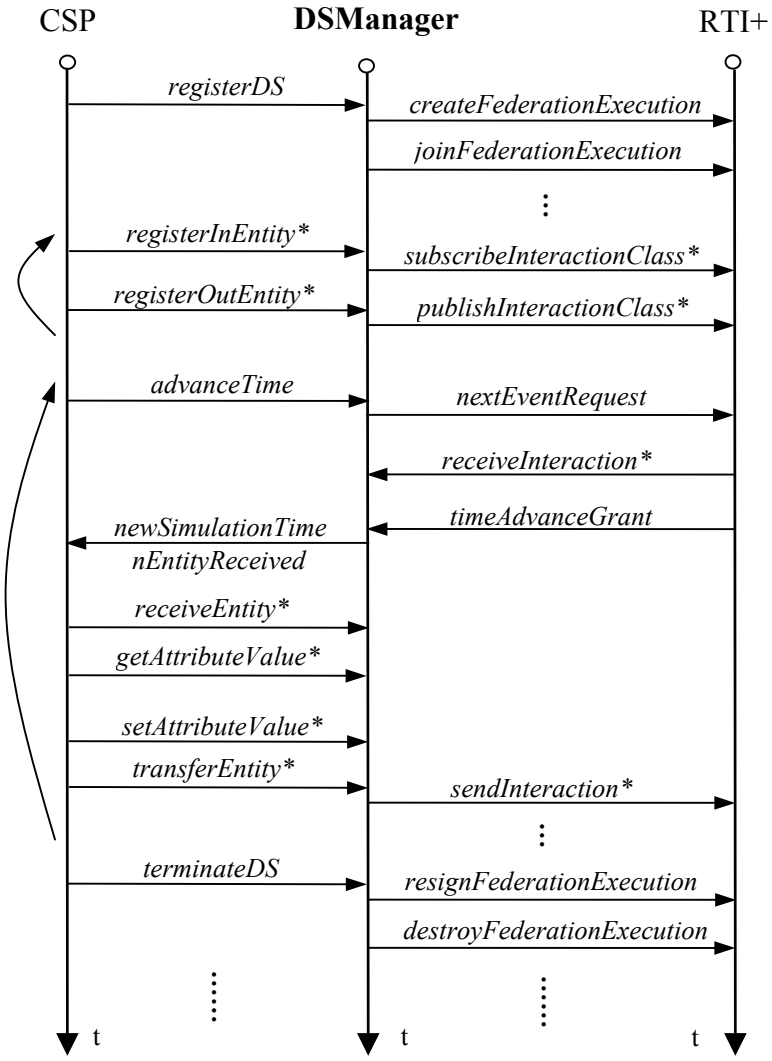


Figure 3.3.1: Interaction among CSP, DSManager and RTI+

During the simulation execution, each model needs to advance time to progress the whole distributed simulation. There are various approaches to time management when using the HLA RTI to support

distributed simulation. The approach described here is based on *NextEventRequest* (conservative synchronization). When the CSP wishes to advance to the time of its next event, it issues an *advanceTime* request to the DSManager. The DSManager invokes the corresponding RTI+ service *nextEventRequest*. The response from the RTI+ is zero or more interactions received via *receiveInteraction* and a new simulation time granted via *timeAdvanceGrant*. The interactions represent the arrival of entities at the time granted by *timeAdvanceGrant* and this time may be less than the time initially requested by the CSP (i.e. if entities arrive before the time of the original next event the new time of the next event is that of the arriving entities). If no interactions appear, the time granted is exactly the requested time. Either way, this grant time is returned as the result of the *timeAdvanceGrant* method to the CSP with the number of entities received *nEntityReceived* (if any). If *nEntityReceived* is larger than zero, the CSP will retrieve all the entities via *receiveEntity* and *getAttributeValue* (if the entity has some attributes). In this way, the CSP advances its local simulation time and continues execution. Conversely, if any entities leave the simulation model, the CSP will send them to the DSManager using *setAttributeValue* and *transferEntity* as many times as appropriate. The DSManager will translate these into interactions and then forward these to the RTI+ by invoking *sendInteraction*. This procedure continues until some terminating condition, such as the simulation end time, is met.

The CSP informs the DSManager that the terminating condition is met via *terminateDS*. Before terminating the federation, the controller model needs to invoke some RTI+ services to achieve synchronization ending point. Then the DSManager invokes the RTI+ service *resignFederationExecution* to leave the distributed simulation. In addition, *destroyFederationExecution* is called by the DSManager of the controller model to destroy the distributed simulation.

Besides the functions mentioned above, there are some other functions supporting the distributed simulation. For example, for benchmarking purposes a lookahead value can be set and the synchronization approach to be used can be declared. A full specification of the generic interface is given in Appendix A.

3.3.2 Entity Transfer Specification

The Entity Transfer Specification (ETS) focuses on a common data exchange format of the entity that has been prepared for transfer. It is assumed that there is some translation mechanism between the CSP and the DSManager to convert to and from the ETS representation. It is also assumed that *time* has been converted into the same units and resolution in both models. The entity is transferred in

the form of a timestamped interaction class, where the timestamp is the time when the entity is transferred. When sending an entity, the DSManager will add the travel time (the number of simulation time units for the entity to be transferred between the two models) to the timestamp of the entity and forward it to the receiving model. The entity is represented as an entityName and zero or more attributes as below.

Entity = {entityName, attributes}*

e.g.

wheel= {"wheel", 4, "black"}

which represents a wheel entity with value 4 for attribute (number, integer) and value "black" for attribute (color, string).

In distributed simulation, the entity is possibly transferred from one model to another model. The model which sends the entity is called the source model while the model which receives the entity is called the destination model. In the ETS, an HLA interaction class with the name containing the necessary information is used to represent the passing of an entity from the source model to the destination model at the RTI level. Usually, the information includes the name of the entity, the name of the source model and the name of the destination model. It is possible that entities with the same type from different source models need to be transferred to different destination models. That is why the names of the source model and destination model must be contained in the name of the interaction class. Therefore, the name of the interaction class is defined as:

entityName + **From** + *sourceModelName* + **To** + *destinationModelName*

where the three names provided by the model are connected by default key words "From" and "To".

Note that the generic interface is designed in a way which tries to hide the detailed information of each model from other models. That is, the model only needs to declare what entities they want to receive or transfer and which other models they want to receive entities from or transfer entities to. All the entities of the same type from the same source model will come to the destination model via the same entry point. Therefore, in the destination model multiple external entry points for the same entity type should be associated with different source models. For this reason, in the ETS no information about specific entry points is included in the interaction class name.

For some special types of models, e.g., a monitoring or visualization model, they may need to collect some information exchanged in the federation. Some super class can be used to represent such an interaction class.

entityName + **From** + *sourceModelName*

which allows any model in the federation to receive the entity from the source model with the specified name.

entityName + **To** + *destinationModelName*

which allows the destination model with the specified name to receive the entity from all the sending models in the federation.

entityName

which allows the entity to be received by all the other models in the federation.

In addition, all the attributes of the entity are defined using the parameters of the interaction class. Table 4.3.1 shows an example of the OMT interaction class structure and parameter table for sending an entity. A model named "WPL" needs to transfer an entity with the name of "wheel" to another model "BAL". As to the "wheel" entity, it has two attributes with the name of "number" and "color" separately. Note that in the above for an actual implementation *sourceModelName* and *destinationModelName* are replaced by the "WPL" and "BAL" and *entityName* is replaced by the "wheel" as appropriate.

Table 3.3.1: Interaction class structure and parameter table for sending entity

HLAinteractionRoot (N)	wheelFromWPLToBAL (P)

Interaction	Parameter	Datatype	Available Dimension	Transportation	Order
wheelFromWPLtoBAL	number	Integer	NA	HLAReliable	timestamp
	color	Array of Character	NA	HLAReliable	timestamp

Note that the interaction class 'HLAinteractionRoot' shall be a superclass of all other interaction classes in a FOM or SOM. And each interaction class in the interaction class structure table shall be followed by information on publishing and subscribing capabilities enclosed in parentheses as:

- *P (Publish)*: The federate is capable of publishing the interaction class.
- *S (Subscribe)*: The federate is capable of subscribing to the interaction class.
- *PS (PublishSubscribe)*: The federate is capable of publishing and subscribing to the interaction class.

- *N (Neither)*: The federate is incapable of either publishing or subscribing to the interaction class.

3.4 Extension to DSManager for CSPI-PDG Type II IRM

While asynchronous entity passing focuses on the general problem of entity transfer, synchronous entity passing (CSPI-PDG Type II IRM) represents another more complicated type of model. In the Type II IRM, the sending model may transfer entities into a bounded queue or a workstation with limited capacity in the receiving model. Thus, entities can be transferred only when the sending model is sure that the destination side is not full (queue) or blocked (workstation). This introduces a synchronous feature into the model, which can be solved by exchanging status information dynamically between the models. In addition, another problem arises from the existence of inter-model simultaneous events as discussed below. It requires the exchange of priority information additionally at run time. The status and priority information is converted into an interaction by the DSManager and sent / received through the RTI+.

3.4.1 Problems of Synchronous Entity Passing

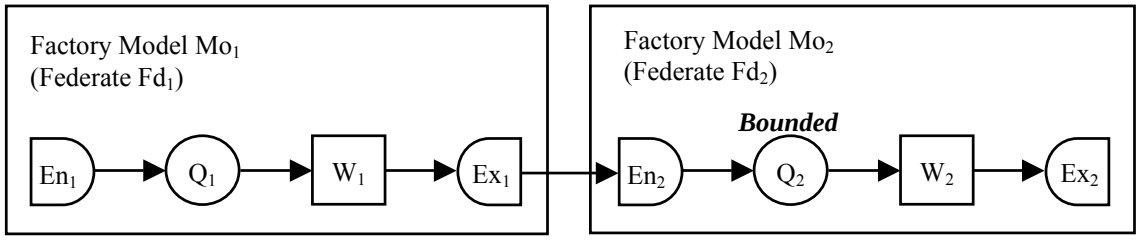


Figure 3.4.1: IRM Type II: synchronous entity passing

The Type II IRM synchronous entity passing deals with the case where a receiving queue is bounded or the receiving workstation has limited capacity. An example is shown in Figure 3.4.1, where the distributed simulation (federation) is composed of two factory models (federates), Mo_1 and Mo_2, interacting in the way denoted by the arrows. Each model consists of an entry point En_i, a queue Q_i, a workstation W_i, a resource R_i, and an exit point Ex_i (where I is the model identifier). After being processed in W_1, entities need to be sent periodically via Ex_1 and then entry point En_2 into a bounded queue Q_2 (or a workstation with limited capacity) in Mo_2. It indicates the requirement that Mo_1 containing the sending workstation W_1 must, when the processing of an entity is complete, check to determine that there is space in Q_2. If there is space available then the entity may be transferred. Otherwise Mo_1 must ensure that W_1 is blocked until space becomes available. As defined in section 2.4.1, the entry point designed to receive entities from external models is called 'external entry point'

with the abbreviation of 'EEP'. On the sending side, the EEP in the receiving model is referred to as the remote EEP, and on the receiving side the EEP is referred to as the local EEP.

3.4.1.1 Inter-model Simultaneous Events

In a discrete event simulation, the events are timestamped and executed in increasing order to ensure causality. It is possible two or more events are scheduled at exactly the same simulation time, or at a slightly different simulation time but below the level of the machine precision. These events are considered as simultaneous events. Different orderings of the simultaneous events may generate different simulation results, which may conflict with the requirement of repeatable execution of the simulation program. Repeatability means the execution of the simulation should produce exactly the same results on each execution when using the same initial state and external inputs.

Much work has been proposed to solve this problem [COR90] [MEH92] [WIE99]. Usually the solution is to execute these events in an arbitrary order unless the modeler explicitly specifies some tie-breaking technique, for example, FIFO (first-in, first-out), LIFO (last-in, first-out), or dependency order. Some tie-breaking mechanisms can be implemented by extending the timestamp to include additional, lower-precision bits that are hidden from the application program [FUJ00]. With different values to these bits, the simulation engine can ensure no two events in the simulation contain exactly the same timestamp. The values could be assigned based on the specified tie-breaking techniques to satisfy the simulation modeler's requirements.

In a standalone simulation, it is relatively easy to order the simultaneous events in the local event list based on some tie-breaking mechanisms. In distributed simulation, however, there may exist some simultaneous events transferred between different model components. For example, two models may want to send entities to the same remote EEP at exactly the same simulation time. These simultaneous events are generated in different models but interleave with each other, referred to as *inter-model simultaneous events*.

Usually, the modeler will assign a priority to order the entities from different sending models. For those cases where no priority is explicitly specified, they are ordered in an arbitrary order. Here the discussion is based on the assumption that the priority is already assigned for each local EEP.

In Figure 3.4.2, the distributed simulation is composed of model M_1, M_2, …, M_n and M_x (n+1 models). Model M_i (I = 1, … , n) generates entities in workstation W_i, and sends them periodically via Ex_i and En_i to a bounded queue Q in Model M_x. In M_x, each local EEP En_i is assigned a different priority for accessing Q. It is possible two or more inter-model simultaneous events exist to transfer entities to Q. In a standalone simulation these entities can be ordered in the event list waiting to be

processed. In distributed simulation the entities from each sending model can be transferred to M_x only when the sending model makes sure there is space available in Q and no entities from other sending models with higher priority need to be transferred to the same queue.

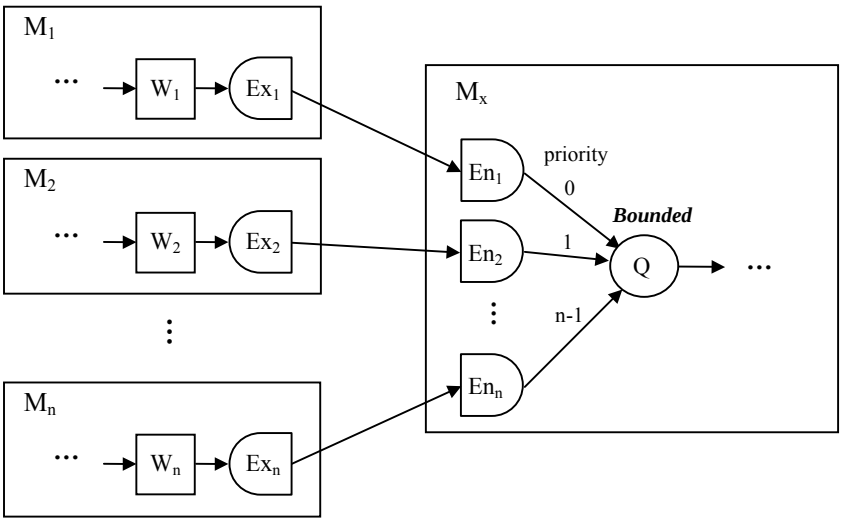


Figure 3.4.2: Inter-model simultaneous events to EEP with single priority

Another case is shown in Figure 3.4.3. M_1, M_2 and M_3 transfer entities into a bounded queue Q or a workstation with limited capacity W in M_x via EEP En_1, En_2 and En_3 respectively. For Q, En_1 has a higher priority than En_2. For W, however, En_2 has a higher priority than En_3. That means one EEP is possibly associated with multiple priorities. For different queues or workstations chosen as the next destination, the EEP will be assigned with a different priority and the entity transferred via this EEP should also be associated with a different priority. Therefore, the priority information should be updated dynamically based on the simulation activities.

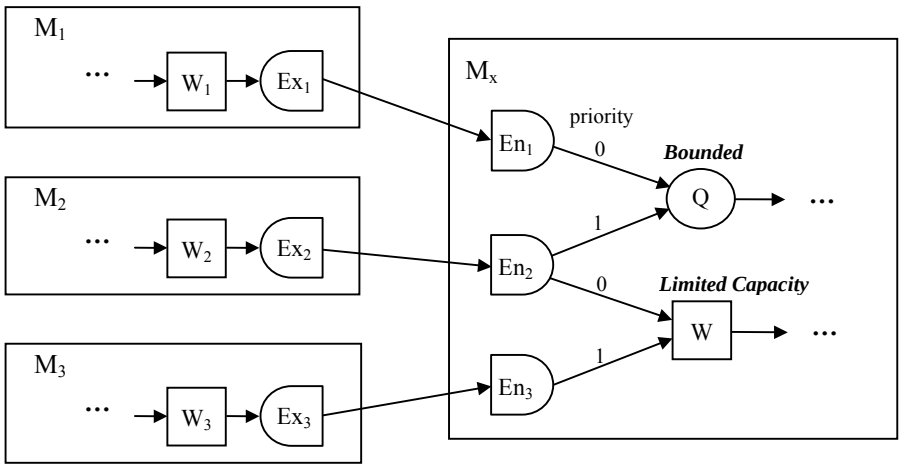


Figure 3.4.3: Inter-model simultaneous events to EEP with multiple priorities

To solve the inter-model simultaneous event problem introduced by synchronous entity passing, solutions are required in the generic architecture. Ideally, the solutions should be implemented transparently to the CSP and the simulation modeler. The CSP for the receiving model needs to

update the priority value to the generic architecture at run time since the model structure is known to the CSP. The CSP for the sending model just needs to check the status of the remote EEP before it transfers entities.

3.4.1.2 Status Information

As discussed above, it is essential to update the status information of a bounded queue or a workstation with limited capacity in synchronous entity passing. Different from simply updating the status information in a standalone simulation, the status information should also be transferred between the models. In Figure 3.4.1, when Q_2 becomes full, a message with a small increment to the current simulation time is sent back to Mo_1, which causes Mo_1 to block. The small increment is added because the status event is dependent on the entity sending event from Mo_1. At some later simulation time, when the entity is processed in workstation W_2, Mo_2 clears a slot in Q_2 and sends another message with a small increment to the current simulation time to Mo_1, which allows new entities to be transferred.

Due to the complexity of distributed simulation scenarios, the status information may not be updated in time to external models. One case is for inter-model simultaneous events. For example, two models may want to send entities to the same remote EEP at the same simulation time. In the situation where there is only space available to receive one entity, the status of the remote EEP cannot be shown as idle for both sending models. Another case is for passing more than one entity with the same simulation time to the same remote EEP from the same sending model. Suppose two entities from Mo_1 are waiting to be transferred into Mo_2 via En_2. After receiving the first entity at time t, it is possible En_2 becomes blocked and the status information of 'blocked' will be transferred to the DSManager in Mo_1 a short time later at $t+\delta$ (δ is the small increment due to the dependency order). However, Mo_1 is trying to send the second entity at t since the new status information can only be received at $t+\delta$. Therefore, even though the receiving model has already updated the status of En_2 as 'blocked' or 'idle' based on local information, the status of the remote EEP may be uncertain for the DSManager in the sending model. The possible status of a remote EEP specified by the DSManager in the sending model can be summarized as follows.

0 : idle and it is safe for the sending model to send an entity

1 : blocked

-1 : uncertain since there are possibly some other entities sent from other models to this remote EEP at the same simulation time

-2 : uncertain since the entity just sent from the local model may cause the remote EEP to be blocked

To avoid the need for the CSP to handle the uncertain status information (the status of a remote EEP known by the CSP is only 'idle' or 'blocked'), the DSManager should update the status automatically and forward 'blocked' to the sending model when the status is uncertain ('-1' or '-2'). After the DSManager is sure it is safe to send an entity from the sending model, 'idle' will be returned instead.

3.4.2 Solutions to Synchronous Entity Passing

To address the new problems introduced by synchronous entity passing, solutions are proposed including extending the DSManager and introducing two hidden fields in the timestamp representation.

3.4.2.1 Extension to DSManager

As mentioned in Section 3.4.1.2, the status information in the Type II IRM is transferred with the timestamp of the current simulation time plus a small increment, considered as a NZL (near zero lookahead) message [WIE98]. Lookahead represents a guarantee from a federate (model) that it will not generate any external message with a timestamp smaller than its current time plus the value of the lookahead. It is critical for conservative synchronization to achieve better performance. In the Type II IRM, however, the lookahead value has to be set to near zero due to the status information. The DSManager will collect information from the model and automatically set the lookahead value. The CSP needs to tell the interface whether each local EEP is restricted or not. Here 'restricted' means the local EEP may be blocked as it is linked to a bounded queue or a workstation with limited capacity. If any one of the local EEPs is restricted, the DSManager has to set the lookahead as near zero. Moreover, if a model needs to send any entity to a restricted remote EEP, the DSManager in this model has also to set the lookahead as near zero. It is because the model may schedule an event at time $t+\delta$ to transfer the entity when the status of the remote EEP is updated from 'blocked' to 'idle' at time t. In other situations, a larger lookahead may be adopted based on the scenario of the model itself.

To handle synchronous entity passing, new functions need be provided to allow the model to update and check status information. In the model which will receive entities from an external model, it must set the status of the local EEP each time it changes. When necessary, it also associates the priority information with the status since it is possible the local EEP has different priorities when it sends entities to different queues or workstations. The local DSManager will transfer such information to the DSManager for the sending model. Before the sending model transfers entities, it will invoke the necessary function to check the status of the appropriate remote EEP. As discussed in

section 3.4.1.2, to hide the complicated implementation details from the CSP and the model, the status returned by the DSManager is only idle or blocked.

3.4.2.2 Hidden Fields in Timestamp Representation

(1) Purposes of Hidden Fields

Hidden fields in the timestamp can be used to solve the problems of simultaneous events. In the DSManager designed for the Type II IRM, hidden fields are utilized for three purposes.

The first purpose of the hidden fields is to represent the small increment to the simulation time for status information. Different CSPs may have different time units and machine precisions in simulation execution. It is difficult to select a suitable value as the smallest time increment. By appending a hidden field of integer type to the simulation time, it can ensure the small increment will not conflict with the timestamp of any event scheduled by the model since the hidden field is transparent to the model layer.

Another purpose of the hidden fields is to contain priority information to order the inter-model simultaneous events. The lower the priority, the larger the value of the hidden field. In this way, the events with higher priority will be associated with a smaller timestamp and will be processed earlier.

The third purpose of the hidden fields is especially for the case when the sending model needs to send more than one entity to the same remote EEP simultaneously, as discussed in section 3.4.1.1. These simultaneous events should be ordered using a hidden field in logical time.

(2) Two Hidden Fields for Synchronous Entity Passing

There are two hidden fields appended to the simulation time to support synchronous entity passing: one is *priority* for priority value to order inter-model simultaneous events (status of '-1), the other is *age* used to order those simultaneous events sent from the same source model to the same remote EEP (status of '-2'). The small increment to the simulation time for status information is also contained in the second hidden field *age*. Thus, the logical time is defined as (*t*, *priority*, *age*) where *t* is the simulation time shown to the model. Importantly, the first hidden field *priority* has precedence (assigned to more significant bits) over the second hidden field *age* (more sensitive). Even for the entities with the same type sent to a remote EEP with a specific *priority*, it is also possible to schedule simultaneous events with different values of *age* (the first entity sent is with *age* 0, the second one is with *age* 1, and so on). It is easier to use two hidden fields to represent the precedence relationship instead of one hidden field.

To ensure the status information is updated as soon as possible, the small increment of simulation time is added to the second hidden field, which is more sensitive than the first one. The value sent for the small increment is less than the value increased each time the model needs to send another entity to the same remote EEP. Here, each age *a* (*a* is a non-negative integer 0, 1, 2, …) is represented as 10**a* (0, 10, 20, …) and 5 (any value between 1 to 9 is acceptable) is used as the small increment in *age* for status information. Consequently, the near zero lookahead discussed previously is also set as 5 in the second hidden field since it is the smallest increment for the logical time.

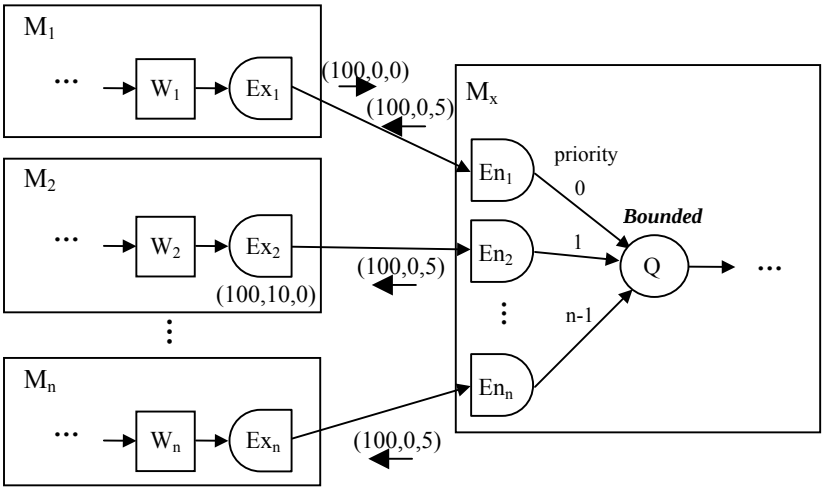


Figure 3.4.4: Time Representation with Hidden Fields

Let us illustrate the hidden fields using the case in Figure 3.4.2. Suppose the status of Q at time *t* is not full. Only M_1 can directly transfer an entity to Q because the corresponding remote EEP En_1 has the priority of 0. For each other sending model M_i (i = 2, 3, … , n) that wants to transfer the a_i^{th} (a_i = 0, 1, …) entity at time *t*, the DSManager sets *priority* as p_i (p_i = 2, 3, ... , n) and age as $10*a_i$, and tries to advance time to (t, p_i, $10*a_i$). Only when the granted time is equal to the requested time and no status information of 'blocked' is received during the time advancement, is the 'idle' status returned to the model by the DSManager for M_i. If the a_i^{th} entity sent from M_i causes Q to become full, the new status information will be sent to all sending models at time (t, p_i, $10*a_i+5$), which stops M_i sending other entities and meanwhile allows the models, including M_{i+1} to M_n, to receive the 'blocked' signal before their requested time is granted. As shown in Figure 3.4.4, suppose both M_1 and M_2 need to send an entity at time *100* and the current status of Q in M_x is idle. The DSManager will modify the model time as (*100*, *0*, *0*) and (*100*, *10*, *0*) respectively after including the priority information of the remote EEP. Therefore, the entity from M_1 will be transferred to Q via En_1 first. Suppose Q is full after this action, then status information will be sent to both M_1 and M_2 at time (*100*, *0*, *5*) since a small increment needs to be added to the time representation in the DSManager. This event will cause En_2 to be blocked in M_x. Because (*100*, *0*, *5*) is earlier than (*100*, *10*, *0*) the status will be received by M_2 before it intends to send an entity at (*100*, *10*, *0*). In this way, the entities from the sending models can be sent in the correct order as specified by the priority of the

corresponding remote EEP. Moreover, the status information can also be updated and received in a timely way.

3.4.3 Implementation Issues

The proposed solutions are implemented in the DSManager. This includes the implementation of a user-defined logical simulation time as allowed by the IEEE HLA standard.

3.4.3.1 The DSManager

The DSManager provides an interface consisting of a set of functions to be invoked by the 04. when a distributed simulation is created. Through the interface, the DSManager invokes necessary calls to the RTIAmbassador on behalf of the 04. and transfers the information received from the FederateAmbassador to the 04.. The basic communication protocol between the 04., DSManager and RTI for 04.I-PDG Type I IRM is described in Section 3.3. Here only the new features in the interface to the 04. for the Ty pe II IRM are discussed.

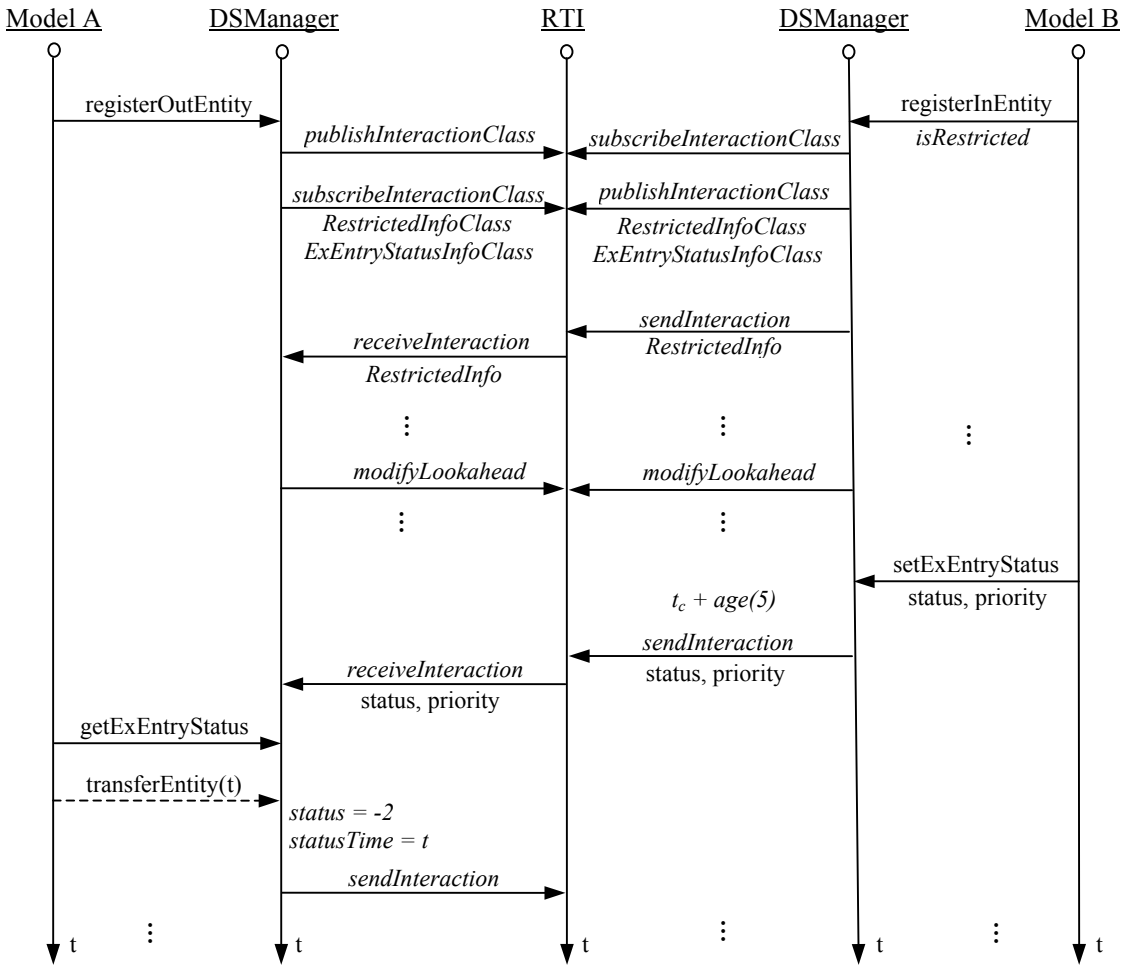


Figure 3.4.5: New features of interface for Type II IRM

In Figure 3.4.5, model A and model B are used to indicate the sending and receiving models respectively. Suppose model A transfers entities to a bounded queue or a workstation with limited capacity in model B. On each side, there is a DSManager used to communicate with the HLA RTI on behalf of the model. In the initialization phase, model A and model B need to register the entity which is exchanged via *registerOutEntity* and *registerInEntity*. It should be noted that the '*RestrictedInfo'* information is also provided by each local EEP in model B. If any local EEP is restricted, the DSManager in Model B will call *modifyLookahead* to modify the lookahead value to near zero. Correspondingly, the lookahead in Model A should also be set to near zero by the DSManager in Model A. This information can be forwarded to the DSManager in model A by invoking *sendInteraction*. Before that, the DSManager on each side needs to declare the interest to send or receive such information by calling *publishInteractionClass* and *subscribeInteractionClass*. Additionally, the DSManager also automatically declares the interest to send or receive the status information as well as the priority for each EEP.

During the simulation execution, if the status of a local EEP is changed due to the simulation activities, model B will inform the DSManager by calling *setExEntryStatus* with the new status (idle or blocked) and current priority. Instead of increasing the time at the model level, the hidden field age is increased by the small increment which is transparent to the model. Then the DSManager will transfer the information to model A via *sendInteraction*.

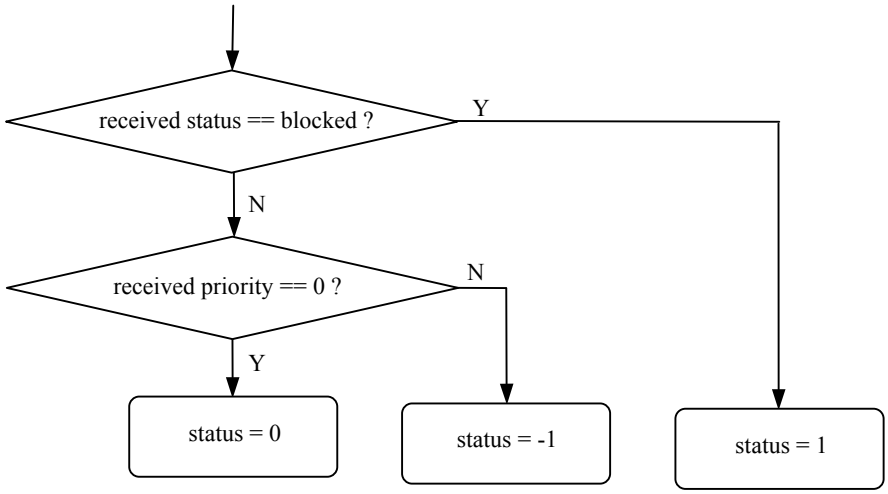


Figure 3.4.6: Receive *ExEntryStatus* procedure

The DSManager for model A will set the status based on the received information (as shown in Figure 3.4.6). If the status is idle while the priority value is larger than 0, it is possible entities may be transferred to a remote EEP with a higher priority which also shares the queue or workstation with the remote EEP for this entity. In this case, the status is uncertain and has be set as '-1'. Before transferring an entity to model B, the CSP needs to check the status of the corresponding remote EEP

in model B using *getExEntryStatus*. The DSManager will return 'idle' or 'blocked' after considering the simulation activities in the local model in addition to the status and priority information received from model B (as shown in Figure 3.4.5). After transferring an entity via *transferEntity* to model B, the DSManager in model A will locally change the status of the corresponding remote EEP to '-2' since the entity may cause the remote EEP in model B to be blocked.

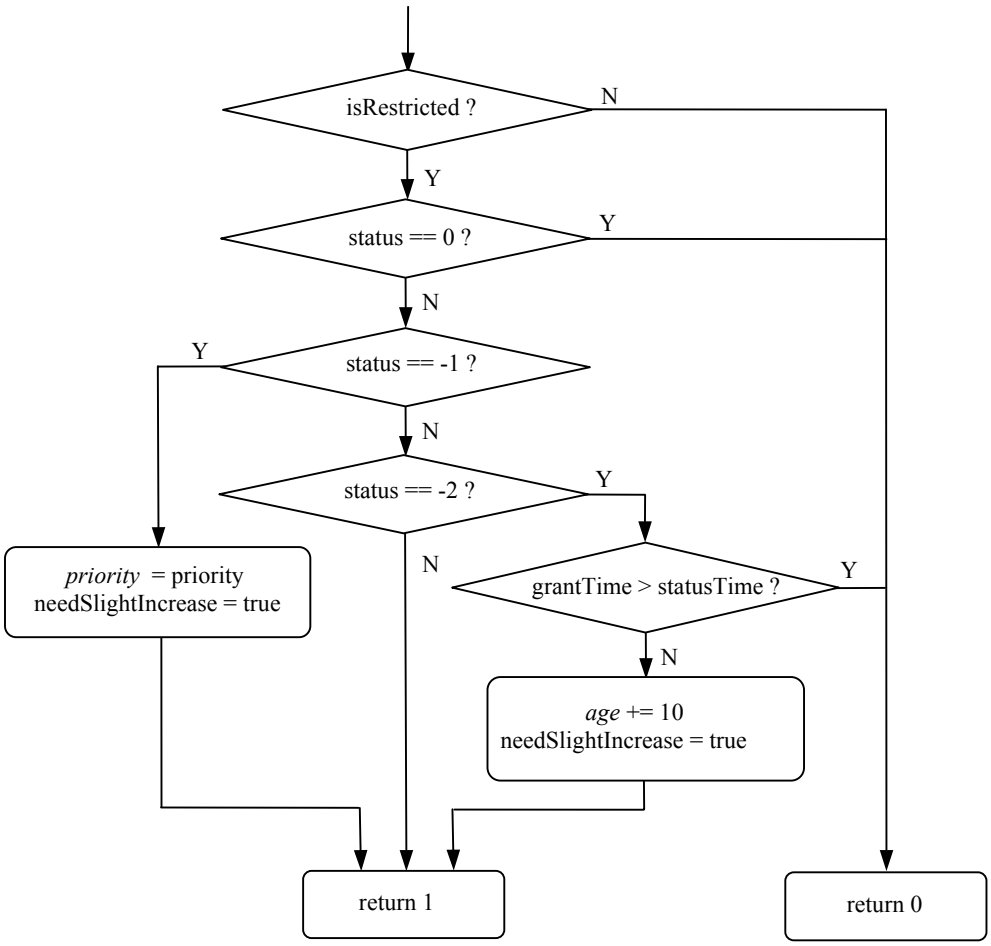


Figure 3.4.7: *getExEntryStatus* procedure

As it is known, each model needs to advance time to progress the whole distributed simulation. Specifically, in the Type II IRM, the requested time forwarded to the RTI is possibly associated with a slight increase represented by the hidden fields. This may slow down the simulation if the hidden fields are added for each time request. The variable '*needSlightIncrease'* is used to identify whether it is necessary to set the hidden fields in the next request to advance time. Figure 3.4.7 shows that it is only set to 'true' when the hidden field needs to be appended. Another variable '*statusTime'* gives the time when the new status is updated. After sending the entity to the external model, the status is set as '-2' and the *statusTime* is updated to the current logical time. However, if the current granted time is larger than *statusTime* and the status is still '-2', this means there is no new status information

of 'blocked' received from the external model. In this case, 'idle' is returned to the model. Otherwise, 'blocked' is returned since the status is still uncertain, and the hidden field age should be increased enough to see whether there is new status information received in the next request to advance time.

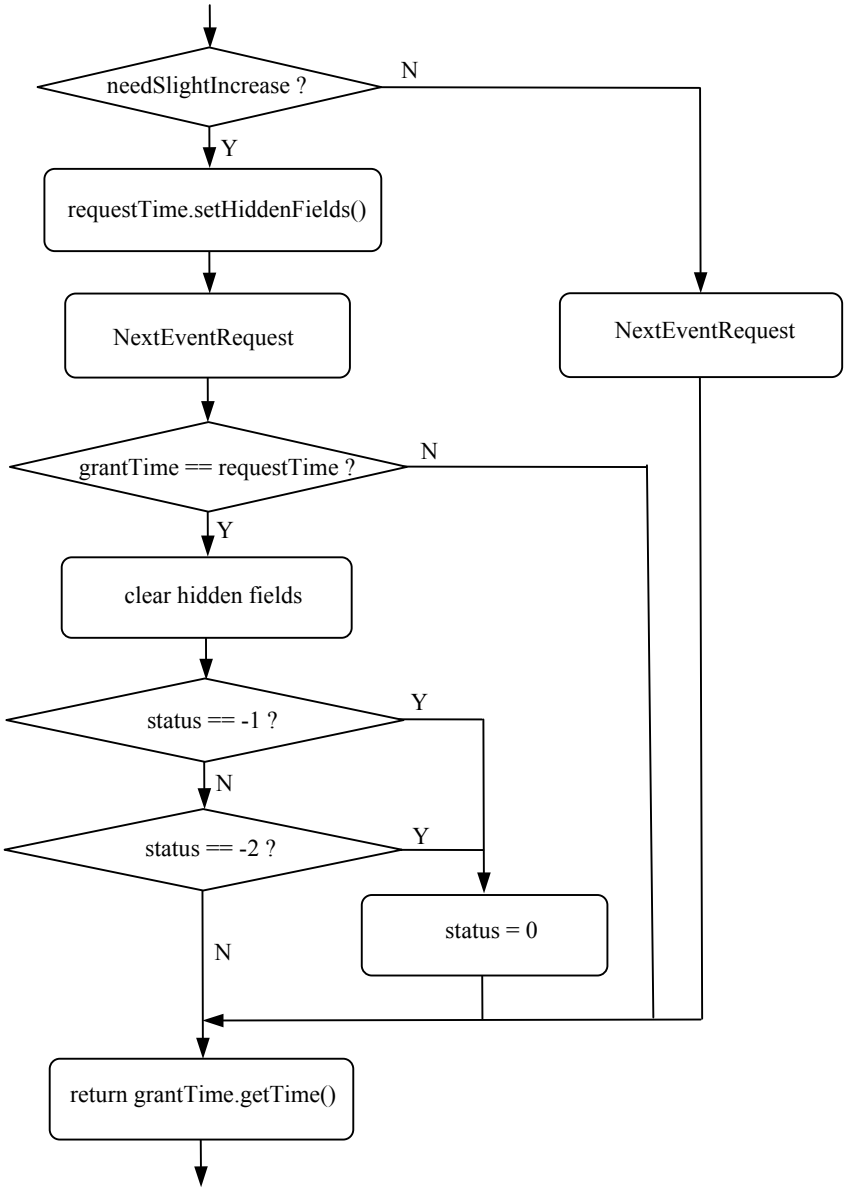


Figure 3.4.8: *advanceTime* procedure

Figure 3.4.8 shows the general procedure for time advancement. Each model advances time by invoking *advanceTime* to the DSManager. In the procedure, the hidden fields may be added to the requested time (requestedTime) provided by the CSP and passed to the RTI (by calling the *setHiddenField* method). After a safe time is granted from the RTI, the DSManager will clear the hidden fields and update the uncertain status information based on the granted time and received status information (if any) during the time advancement. If the granted time is equal to requested

time and the status is still '-1' or '-2', the status is updated to '0' since that means no new status information of 'blocked' is received before the requested time during the time advancement. Finally, the simulation time without hidden fields (obtained by calling *getTime* method) is returned to the model since the hidden fields are transparent to the model.

3.4.3.2 FedTime in the RTI Implementation

In the HLA standard, logical time is defined as an abstract class which allows the user to implement a version of this class for their own purposes. This provides the possibility to add the hidden fields to the *FedTime* in the RTI implementation while still conforming to the HLA standard. To extend *FedTime* with the new attributes of '*priority*' and '*age*', some supported functions are provided for operation on and comparison between logical time values. For instance, for comparison using the '>' operator, suppose there are two timestamps: T_1 (t_1, $priority_1$, age_1) and T_2 (t_2, $priority_2$, age_2). If t_1 is larger than t_2, the result is 'true'; else if t_1 is equal to the t_2, the result is 'true' when $priority_1$ is also larger than $priority_2$; else if t_1 is equal to t_2 and $priority_1$ is equal to $priority_2$, the result it 'true' only when age_1 is also larger than age_2. Modifications also need to be made to the *encode* and *decode* functions in the *FedTime* class to include and exchange the hidden fields via the network.

In this implementation, the *FedTime* class provided by DMSO RTI1.3NG-V6 [RTI02] is extended and the new generated library *libFedTime* is linked to the DSManager.

3.4.4 Entity Transfer Specification

Based on the entity transfer specification discussed for Type I IRM, special consideration should be made for Type II IRM. In addition to the entities exchanged between the models, some other HLA interactions should be added to contain more information necessary for synchronous entity passing.

One interaction is used for transferring *RestrictedInfo* information in the initialization phrase. The name of the interaction class is defined as:

entityName + **RestriInfo** + **From** + *destinationModelName* + **To** + *sourceModelName*

Note that *RestrictedInfo* information is provided by the destination model which will receive the entity from the source model. In terms of *RestrictedInfo* information, the DSManager will decide whether the lookahead of the model should be set to near zero. Table 3.4.1 shows an example of the OMT interaction class structure and parameter table for exchanging *RestrictedInfo* information. Suppose model WPL will send entities to model BAL, then BAL needs to tell the DSManager

whether the entities will be received by a restricted simulation object (a bounded queue or a workstation with limited capacity).

Table 3.4.1: Interaction class structure and parameter table for *RestrictedInfo* information

HLAinteractionRoot (N)	wheelRestriInfoFromBALToWPL (P)

Interaction	Parameter	Datatype	Available Dimension	Transportation	Order
wheelRestriInfoFromBALtoWPL	RestrictedInfo	Boolean	NA	HLAReliable	receive

The other HLA interaction is used to update the status and priority information during the simulation execution. The name of the interaction class is defined as:

entityName + **EnStatus** + **From** + *destinationModelName* + **To** + *sourceModelName*

Such information is also provided by the destination model which will receive the entities from the source model. Table 3.4.2 shows an example of the OMT interaction class structure and parameter table for exchanging *Status* and *Priority* information. Similar to the example in Table 3.4.1, model WPL will send entities to model BAL. If the entities will be sent to a bounded queue or a workstation with limited capacity, BAL needs to update the status information as well as possibly changed priority information to WPL to block or unblock new entity transmission.

Table 3.4.2: Interaction class structure and parameter table for *status* and *priority* information

HLAinteractionRoot (N)	wheelEnStatusFromBALToWPL (P)

Interaction	Parameter	Datatype	Available Dimension	Transportation	Order
wheelEnStatusFromBALtoWPL	Status	Integer	NA	HLAReliable	timestamp
	Priority	Integer	NA	HLAReliable	timestamp

3.5 Summary

In this chapter a generic architecture is proposed for the integration of the CSP with the HLA. It adopts an implicit approach from the modeler's point of view. This is realized by the cooperation between the simulation model, the CSP and a middleware which includes a DSManager that handles data exchange and an extended RTI (RTI+) that implements the synchronization algorithm. With the generic architecture, each CSP can provide the functionality to satisfy the requirements listed in section 2.3.1 and then join a large-scale distributed simulation, even geographically dispersed. In addition to the conservative synchronization approach, the optimistic synchronization approach will also be supported in the generic architecture. This allows the modeler to choose a suitable approach according to the characteristics of the specific application.

To free the CSP and the modeler from the requirements of knowing the HLA and distributed simulation as much as possible, the DSManager provides a generic interface supporting entity transfer. The interface is made up of a set of C functions, which are easy to understand and invoke for the CSP. In some CSPs it is possible to hide the DSManager calls inside the interface of the CSP, thus providing a truly implicit approach. In some other CSPs, however, this may not be possible, and the modeler may have to put DSManger calls into the code manually. Currently, the DSManager supports the CSPI-PDG Type I IRM and Type II IRM. The DSManager for Type I IRM focuses on the general interoperability problem, including entity transfer, information exchange, and time management. Due to the new problems generated for Type II IRM synchronous entity passing, some modifications are made to the DSManager as well as the underlying HLA RTI time representation. A good feature is that the DSManager for Type II IRMs also supports Type I IRMs. In this way, the model only needs to inform the DSManager whether each local EEP is restricted (linked with a bounded queue or a workstation with limited capacity) or not, without identifying the type of the model itself. The interface will be verified in the next chapter based on some experiments.

CHAPTER

4

THE CSP EMULATOR AND EVALUATION

4.1 Introduction

In Chapter 3, the generic architecture for integrating CSPs with the HLA is proposed and the DSManager is developed to facilitate the CSP to communicate with the HLA. However, a challenge arises from the implementation of the proposed generic architecture. Due to the heterogeneous properties and various degrees of extensibility of CSPs, it is extremely difficult to find a general approach for the integration of CSPs with the HLA. Any experimental integration of a specific CSP with the HLA only favors this CSP and may not be easily applied to other CSPs. After surveying some of the currently popular CSPs, Simul8, Witness and Arena, a CSP Emulator (CSPE) is proposed that includes general features of CSPs and emulates the functionality and interface to a CSP. It can be used to investigate and to benchmark alternative interoperability solutions. The CSPE supports the creation of a standalone model in the same way as CSPs, and additionally it has some new features used by the modeler to build a component model as part of a distributed simulation.

To enable a model built using a CSP to join a distributed simulation, some new features should be developed for the CSP, including the interface used by the modeler and the internal simulation engine of the CSP itself. Here the simulation engine refers to the control system or simulation executive which is responsible for sequencing the operation of simulation activities which occur as the simulation proceeds. Some work has been done in this area to suggest new features to be added to Simul8 in particular and CSPs in general to provide interoperability functionality [RYT03]. Similarly the CSPE described in this thesis has some special features for this purpose. This CSPE is more general than the CSPE in [TTM05], which was first introduced to create a pipeline model for benchmarking purposes. The difference includes increased functionality and a more flexible user interface for building general models.

4.2 Requirements for the CSPE

4.2.1 Survey of Standalone CSPs

The design of the CSPE is based on the investigation of three popular CSPs, Simul8, Witness and Arena. Generally, a CSP supports the creation of simulation models using graphical model construction (icon or drag & drop) or programming / access to programmable modules. In addition to the model design, a basic simulation engine must also be provided to execute the simulations. The focus here is on those CSPs which can support discrete event simulation. The simulation engine should be able to schedule and process the events, and subsequently update the states if they are changed by these events.

Usually, the simulation model is constructed using entities and logical relationships which link different entities together. Entities are similar to tangible obects in the real world. For instance, in a manufacturing system, an entity could be a machine or a vehicle, or a wafer being processed in a machine. Generally, an entity could be temporary (a part that passes through the model) or permanent (a machine remaining in the model) [BAL06]. The logical relationships link different entities together, for example, a part entity needs to be processed by a machine entity. In general, the simulation is executed by moving the temporary entities through the model via permanent entities. The routing of the temporary entities is often defined by the connections among different permanent entities. Each CSP has its own terminology and approach to describe the entities and their connections. Table 4.2.1 shows how entities, temporary entities, permanent entities and connections are represented in Simul8, Witness and Arena.

Table 4.2.1: Model Design in Simul8, Witness and Arena CSPs

	Simul8	**Witness**	**Arena**
Entities	Objects	Physical Elements	Data Module
Temporary Entities	Work Items	Parts	Entities
Permanent Entities	Work Centers Storage Bins (Queues) Resources Conveyors Work Entry Points Work Exit Points	Machines Buffers Labor Vehicles Tracks Fluids Processors Tanks Pipes	Stations Queues Storages Resources Statistics Sets Sequences Conveyors Transporters
Connections	Routing In Routing Out	Input Rules Output Rules	Flowchart module Leave

As mentioned earlier, the CSPs require some kind of simulation engine (control program or simulation executive) to manage the activities of the entities and trace the life history of each entity. There are at least four widely used approaches used for modeling a discrete event simulation as follows [PID04]:

- The activity approach
- The event approach
- The process interaction approach
- The three-phase approach

Each of these approaches embodies a distinctive world view. Among them, the activity approach is easier from the programming viewpoint since its focus is on the relationships between the activities following from specific conditions during the simulation. But it treats each activity as independent and significant time is spent in the activity scan, which leads to run-time inefficiency. By contrast, the event approach can run faster because it involves execution of only those events scheduled in the future event list. For the process interaction approach, its basic building block is the whole process of an entity, which is a mapping of the life history of each class of entity. Here the process is defined as the sequence of operations through which an entity must pass during its life within the system [PID04]. However, it requires a rather complex simulation executive and interacting processes are more complex to program than activities. Based on the above approaches, the three-phase approach was proposed, which combines the simplicity of the activity approach with the efficient execution of the event approach. It is adopted by many of the current CSPs.

4.2.2 New Features for Distributed CSPs

To support distributed simulation, the CSPs need to be extended to interact with different model components running in other CSPs. There are two main areas for extension. The first is to provide a user-friendly interface for the modeler to declare the interoperability information. For instance, a local model should define what types of entities it wants to send or receive from other models, from which model the entities come or to which model the entities will be transferred. The second area is that the CSPs need to communicate with middleware such as the DSManager to exchange interoperability information. For instance, each CSP should be aware of which distributed simulation it will join, it should have a time synchronization mechanisms to coordinate its local model with other model components during simulation execution, and it may receive entities from and transfer entities to other model components. All these can be done by invoking corresponding services provided by the DSManager. Together with the generic architecture and the DSManager middleware,

the CSPE acts as an example of a standardized approach for a standalone CSP to be extended to support distributed simulation.

4.3 Structure of the CSPE

This section introduces the general structure of the CSPE. After presenting the main elements to support model design, the internal simulation engine to control the simulation execution is described. Among several approaches for the simulation engine, the three phase approach is chosen in the CSPE. In addition to those features to support standalone simulation, some other special features are designed to support the interoperability functionality in the CSPE. A model built using the CSPE may become part of a distributed simulation by providing some necessary information to the CSPE (as discussed in Section 3.2.2). It is achieved through the new features introduced in the CSPE for supporting distributed simulation. These suggested external and internal features provide an example of how current CSPs may be enhanced with interoperability functionality.

4.3.1 Features of the Standalone CSPE

4.3.1.1 Model Design

Table 4.3.1: Model Design in CSPE

	CSPE
Entities	Objects
Temporary Entities	Entities
Permanent Entities	Work Stations Queues Resources Entry Points Exit Points
Connections	Routing In Routing Out

Based on the survey of three popular CSPs, the CSPE is developed to have simple but similar functionality to support standalone simulaton. Table 4.3.1 shows the main elements in CSPE to support model design. Specifically, the temporary entities have some attributes associated with them to define their properties. For permanent entities, different types have different properties, for example, a work station has capacity, processing time, breakdown definition and routing in / out rules, while an entry point has entity type (an entity will enter the model through the entry point), arrival interval, and routing out rule. In CSPE, the routing in rule includes priority, FIFO, passive,

collecting and batch, while the routing out rule includes circulate, uniform, percent, priority, split and entity attribute.

4.3.1.2 Simulation Engine – Three Phase Approach

The three phase approach used for modeling a discrete event simulation was first suggested by Tocher [TOC63] and is discussed in detail in [PID04]. The event approach is based on the notion that all future activity can be controlled directly by the simulation engine. On the other hand, the activity approach assumes the simulation engine has no idea about which particular activity is due to be processed at any simulation time. Therefore, it has to spend a lot time to scan all the activities until it finds those which are due to be processed. The three phase simulation distinguishes between those activities which can be directly scheduled and those which cannot. It is achieved by defining two different types of system event (activity).

- **Bound (B) event** These events can be scheduled in advance, bound to happen at some time. For example, in a manufacturing system, the end of service in a workstation can be considered as a B event.

- **Conditional (C) event** These events cannot be scheduled to occur at some time in advance. They depend on some condition, i.e. the states of system resources or availability of the entities. For example, in a manufacturing system, the start of service in a workstation happens only when the workstation is idle and there is some entity waiting for the service.

A three phase simulation has three phases, called A, B and C phases (shown in Figure 4.3.1). These three phases are executed continually in a cycle as the simulation proceeds. The phases are as follows:

- **A phase (time scan):** During this phase, the simulation executive determines when the next event is due to happen and then advances the simulation time to this next event time.

- **B phase (B events execution):** During this phase, the simulation executive will process all the B events which are due to execute at the current simulation time. After processing these events, some other new B events or C events may be generated. In addition, these B events may change some conditions (e.g. the states of workstations) and lead to some C events to be executed at the current simulation time.

- **C phase (C events scan and execution):** During this phase, the executive will scan all the C events and execute those C events whose conditions have been satisfied at the current simulation time. Processing C events may also change some states which may satisfy the

conditions of other C events. This is repeated until no more C events can be executed in this phase.

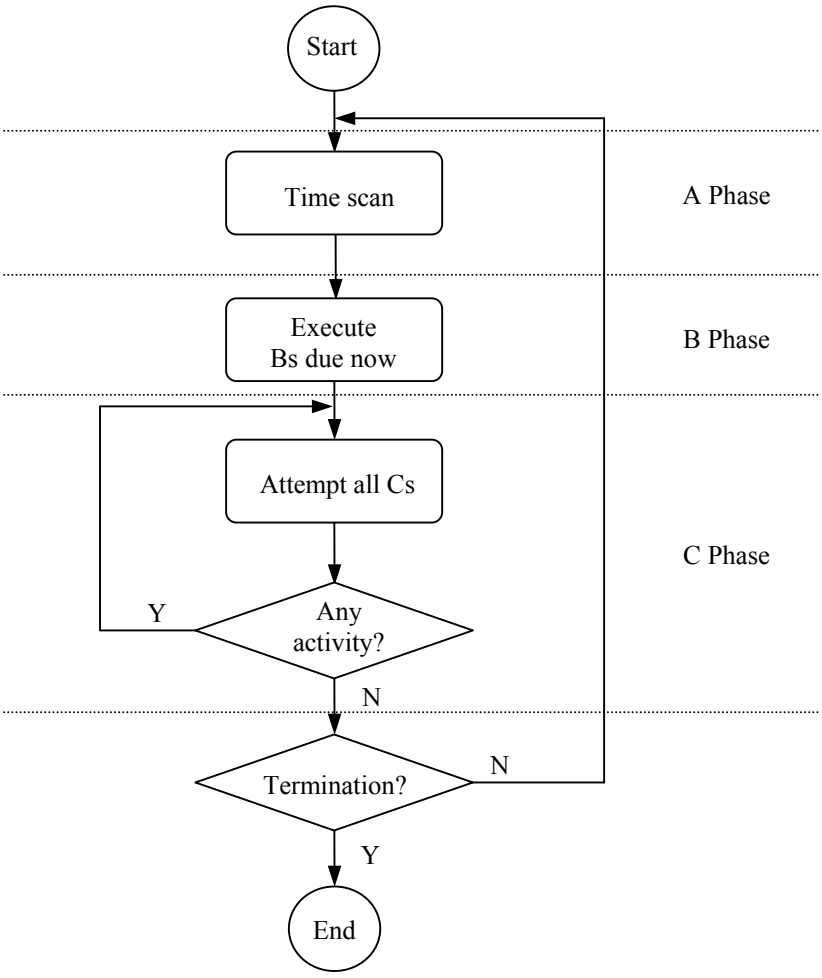


Figure 4.3.1: The three-phase approach

In the CPSE, the three phases are executed with operations on two event lists as follows:

- **Bound (B) event list:** The list of B events, referred to as the B event list, is ordered by the timestamp of the B event which indicates when the event will happen. For simultaneous events with the same timestamp, they are ordered by some other elements, such as, the priority of the event or the priority of the entities themselves.

- **Conditional (C) event list:** The list of C events, referred to as the C event list, is also ordered by some rules, such as the time when the entity starts waiting for the service, the priority of the event or the priority of the entities themselves.

One important issue in three phase simulation is the C event schedule. In the C phase, all the C events are scanned, maybe many times. If the C event list is very long, much time will be spent on

the C phase, which leads to inefficient simulation. Therefore the C event list should be kept as short as possible.

In a manufacturing simulation system, there are usually some entities in a queue, waiting to be processed in a workstation. It is usual to schedule this kind of activity as a C event waiting for the condition that the workstation becomes idle. Here this type of C event is called a *WaitingWS* event. The entity information is associated with the corresponding C event. However, in some situations where the entity arrival interval is less than the processing time of the workstation, the queue will keep increasing and in turn the C event list will become quite long, resulting in a time consuming C phase. An alternative way is not to produce a C event for this kind of activity. Instead the entity information is associated with the queue where the entity is stored. If any workstation becomes idle, all of its input queues will be scanned to find possible entities waiting to be processed by the workstation. In this approach, however, some unnecessary time will be spent in checking the queues without entities in them.

An approach is proposed which tries to exploit the advantages of the previous two approaches. That is, one and only one C event with type *WaitingWS* is scheduled for each queue with entities in it. The information about each entity in the queue is also kept in the corresponding queue object. To make sure that there is at most one *WaitingWS* event for each queue, the executive will check the number of entities in the queue before scheduling the C event. If the number is 1 after putting the entity in the queue, a *WaitingWS* event will be scheduled otherwise if the number is larger than 1 there should already be a *WaitingWS* event for this queue kept in the C event list. After one *WaitingWS* event is processed, another new *WaitingWS* event is scheduled for the next entity in the queue if the queue is not empty. In this way, less time is spent on scanning C events compared to the first approach. Moreover, it is also more efficient than the second approach since queues without entities will not be scanned when the workstations become idle.

4.3.2 Features to Support Distributed Simulation

4.3.2.1 External Features

The CSPE enables the modeler to design interoperating models through a GUI or a specified file. In the CSPE, an entity is processed and passes through some simulation objects. There are four basic types of simulation objects: entry point, queue (bin or storage), workstation (machine) and exit point. The modeler needs to define the attributes and property of each entity type, and assign and link

necessary simulation objects to process each entity. To be part of a distributed simulation, some additional menu options are provided in the CSPE.

Figure 4.3.2 shows part of the menu for entity definition. The entity can be generated in the local model or received from some external models, and can remain in the local model or be transferred to some external models. If the modeler chooses that the entity is exchanged with external models, the name of the source models or destination models should be input in the list.

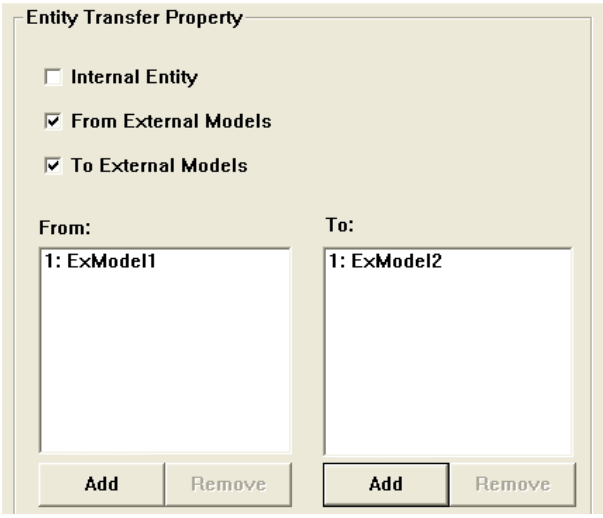


Figure 4.3.2: External entity

Figure 4.3.3 shows part of the menu for entry definition. Choosing an external entry means the entity comes from an external model. It should be noted that each external entry here is only for one type of entity from one source model. The modeler can use other external entries if there are entities from other source models.

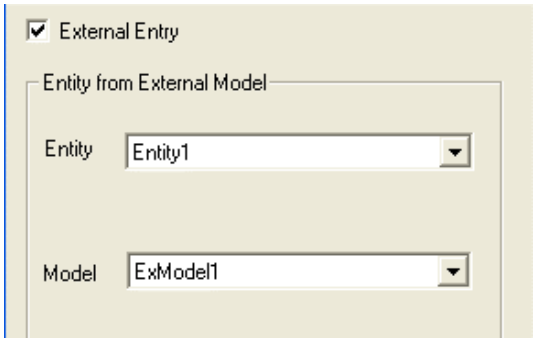


Figure 4.3.3: External entry

Figure 4.3.4 shows part of the menu for exit definition. Similarly to external entry, choosing an external exit means the entity finishing its life cycle in the local model will be transferred to an external model. It is possible there is more than one entity that needs to be sent to the same destination model at the same time (one kind of simultaneous events). Some tie-breaking rule is

needed to schedule the ordering of these simultaneous events. In the CSPE, the approach used is to assign a different priority to each external exit as described in Section 3.4.1.1. The higher the priority (lower value), the earlier (in real time) the entity will be sent out.

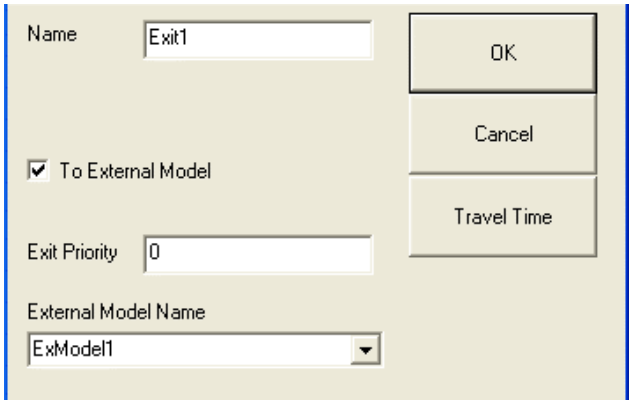


Figure 4.3.4: External exit

Figure 4.3.5 shows part of the menu for the general definition of the model. A model built using CSPE can be a standalone model or part of a distributed simulation. The modeler needs to choose one and only one of the component models as a controller model. The controller model is in charge of managing the creation and termination of the distributed simulation. The number of component models in the distributed simulation is only needed by the controller model. It is used for the initialization phase of the simulation execution. Each component model also should give the name of the distributed simulation, and the name of the FED/FDD configuration file which is used to supply the RTI with all necessary federation execution details during the creation of a new federation [DOD98][IEEE1516]. The menu also shows the names of all external models interoperating with the local model.

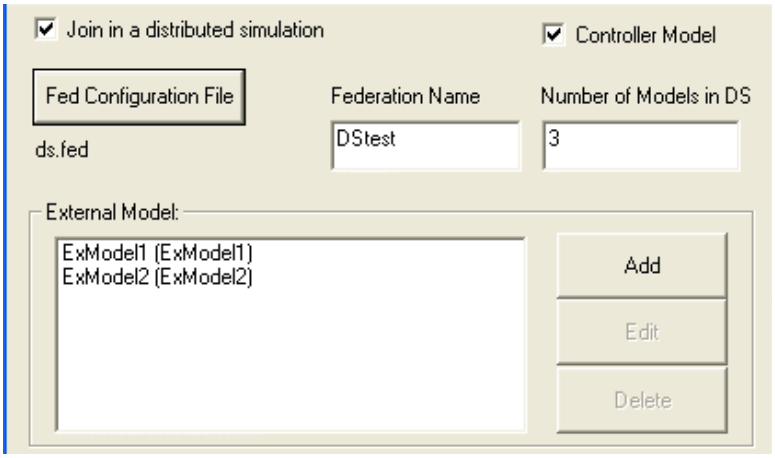


Figure 4.3.5: Component model in a distributed simulation

By providing the above additional information in a straightforward way from the modeler's point of view, the model built using the CSPE can join a distributed simulation. The modeler need not worry about the details of transferring entities among the models and is able to build the models easily and efficiently.

4.3.2.2 Internal Features

In addition to managing the execution of the local model, the CSPE is responsible for interacting with the DSManager by invoking the functions of the interface described in Section 3.3.1. In the initialization phase, the CSPE registers its model as part of a distributed simulation if it is not a standalone model. Then it needs to forward necessary interoperability information of the distributed simulation to the DSManager, such as the name of the local model, the name of the distributed simulation, the name of the FED/FDD file, time policy, and lookahead. It also needs to tell the DSManager what entities its model will exchange with other models in the distributed simulation. During the simulation execution, the CSPE needs to advance the simulation time by providing the time of its next event. As a result of time advancement, it may receive entities with attributes sent from other models. The CSPE then searches the appropriate external entry point from which each entity will be received and schedules a new entity arrival event with associated timestamp in the future event list. Conversely, if any entity leaves the local model via an external exit point, the CSPE will send it to the receiving model by invoking corresponding functions of the DSManager. When some terminating condition is met, e.g., the simulation end time, the CSPE will inform the DSManager that its model has finished its own simulation tasks and wishes to leave the distributed simulation. After receiving a reply from the DSManager, the CSPE can finally terminate the simulation execution of its local model.

For CSPI-PDG Type II IRMs, the CSPE has extra tasks as described in Section 3.4. For example, it needs to update the status of the external entry points in the local model when they are changed, or check the status of external exit points related with the destination model as appropriate. More work is also necessary for handling simultaneous events received from different external models. The CSPE may also need to provide some tie-breaking rule (e.g. priority information) to the DSManager to achieve simulation consistency over repeated executions.

4.4 Evaluation and Experimental Results

Using typical models of CSPI-PDG Type I and Type II IRMs, experiments are conducted to evaluate the generic architecture, in particular: 1) to evaluate the correctness of the CSPE and the interface for

Type I and Type II IRMs; 2) to compare the performance between standalone and distributed simulation varying experimental factors such as lookahead value and event granularity.

4.4.1 CSPI-PDG Type I IRM

4.4.1.1 Evaluation of the Correctness of the CSPE and Interface

In order to evaluate the correctness of the CSPE, the simulation results between the CSPE and Simul8 [SIMUL8] are compared. Simul8 is one of the popular discrete event CSPs, and is used in almost every industry for a wide variety of applications. Since Simul8 currently cannot support distributed simulation, only a standalone model is built using it. For CSPE, however, both a standalone and distributed simulation are created. The experiments are carried out using a bicycle manufacturing system, which for the distributed simulation is a CSPI-PDG Type I IRM.

Figure 4.4.1 shows a distributed and deterministic simulation for the bicycle manufacturing system. It consists of three main parts: a wheel production line (WPL), a frame production line (FPL), and a bicycle assembly line (BAL) that assembles two wheels to one frame to produce a bicycle. The BAL checks wheels for faults and can return them to the WPL for re-machining (an example of valid feedback for Type I IRMs). To achieve a deterministic model for evaluation, the *Circulate* routing-out rule is used here at workstation W_{3a}. This means that the first entity will go to the first destination (exit point Ex_{3b}), the second work item to the second (queue Q_{3b}) and so on. Frames have no such feedback.

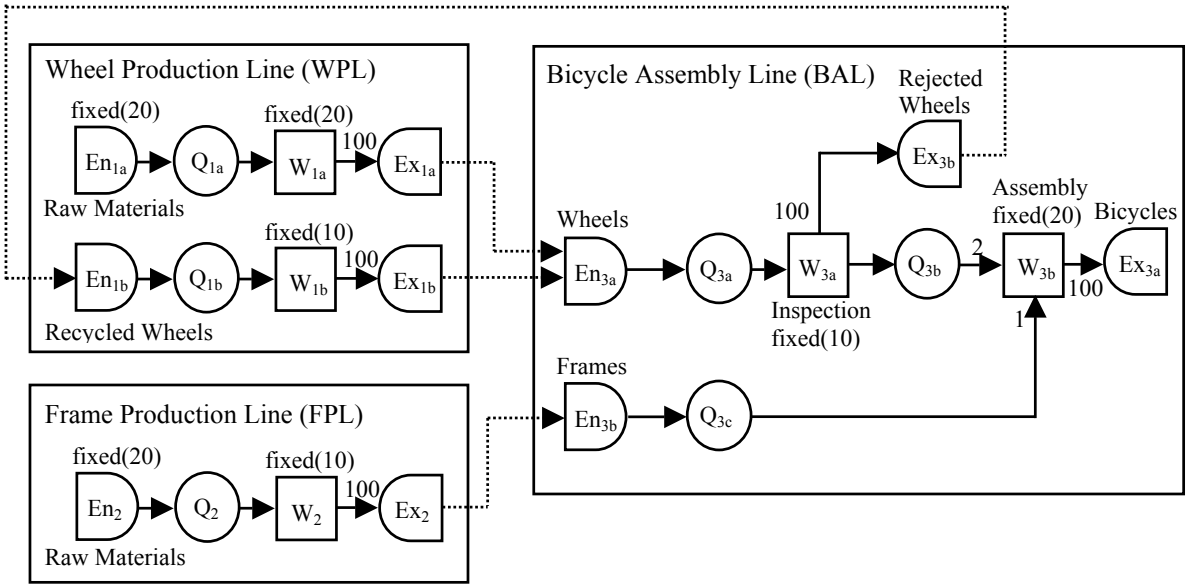


Figure 4.4.1: The bicycle manufacturing system (distributed & deterministic)

Part of the distributed simulation can be described as follows. Raw materials for the WPL arrive every 20 minutes at entry point En_{1a} and wait for processing in Q_{1a}. When machine W_{1a} becomes free, raw materials are taken from queue Q_{1a}, are processed into wheels and released. This activity takes a fixed time of 20 minutes. It is assumed that the entry point, the queue and the machine are adjacent. The newly created wheels then take 100 minutes travel time to be transferred to the BAL's entry point En_{3a}. Note that in the CSPE the inter-model travel time is defined in the sending model. The external exit point in the sending model can be treated as a local representation of the corresponding external entry point in the receiving model. Therefore, the inter-model travel time can be locally defined as the simulation time needed to transfer the entity to an external exit point. This has the advantage that, at any time, an entity is associated with a particular model.

The rest of the distributed simulation can be described in a similar manner with the various times to perform actions shown on the models. In this deterministic model all distributions are fixed, that is, there is no randomness involved in the model. A corresponding standalone and deterministic model is shown in Figure 4.4.2, where the simulation process is the same as the distributed one except that all the process is completed in one combined model named Bicycle Manufacturing System (BMS).

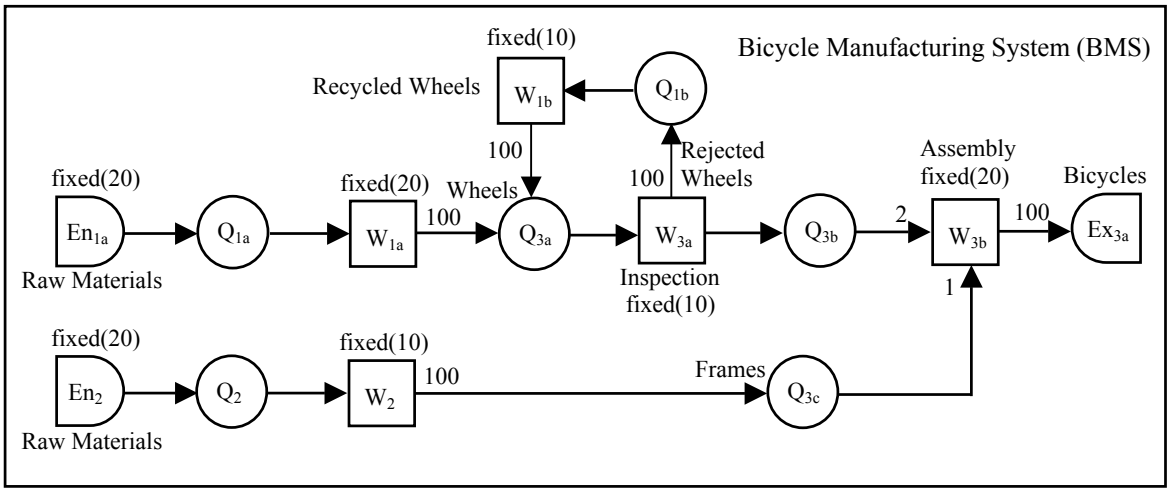


Figure 4.4.2: The bicycle manufacturing system (standalone & deterministic)

The experiments for the distributed simulation are run on four DELL 2.8GHz P4 1GB memory computers connected via a 1Gbps network. One computer is used to run the rtiexec (DMSO RTI1.3NG-V6), the initialization process for the RTI, and the other three for three separate components models (WPL, FPL and BAL models respectively). The experiments for the standalone model are run on one such computer.

Table 4.4.1 shows the experimental results for simulating the system for 100,000 simulation time in Simul8, CSPE(SA) (standalone model) and CSPE(DS) (distributed simulation). The final throughput

of the system as well as the statistics for each simulation object are identical for all three cases, showing the correctness of the CSPE and successful interoperability of CSPE models.

Table 4.4.1: Experiment results for deterministic model

		Simul8	CSPE(SA)	CSPE(DS)
En_{1a}	Arrival Entities	5000	5000	5000
En_2		5000	5000	5000
En_{1a}	Refused Entities	0	0	0
En_2		0	0	0
Q_{1a}	Total Entered Entities	5000	5000	5000
Q_{1b}		4978	4978	4978
Q_2		5000	5000	5000
Q_{3a}		9966	9966	9966
Q_{3b}		4982	4982	4982
Q_{3c}		4994	4994	4994
Q_{1a}	Queue Length at End Time	0	0	0
Q_{1b}		0	0	0
Q_2		0	0	0
Q_{3a}		0	0	0
Q_{3b}		0	0	0
Q_{3c}		2503	2503	2503
W_{1a}	Completed Entities	4999	4999	4999
W_{1b}		4977	4977	4977
W_2		4999	4999	4999
W_{3a}		9965	9965	9965
W_{3b}		2490	2490	2490
W_{1a}	Status at End Time	busy	busy	busy
W_{1b}		busy	busy	busy
W_2		busy	busy	busy
W_{3a}		busy	busy	busy
W_{3b}		busy	busy	busy
Ex_1	Completed Entities	2488	2488	2488

In the above experiments, the distribution used in the model is fixed. However, there are a lot of stochastic elements in 'real' discrete event simulations. The modeler usually uses some kind of distribution to produce a range of values appropriate to the model. The stochastic value is derived from a probabilistic distribution sampled from a random number stream. The range of the values is dependent on the parameters of the probability distribution, e.g., mean and standard deviation. By introducing some probability distributions into the bicycle manufacturing system, another set of experiments is carried out for stochastic models. Instead of a fixed distribution, a normal distribution is used for the processing time in all workstations. For instance, the processing time of W_{1a} is changed from *Fixed* (20) to *Normal* (20, 5) and that of W_{1b} is changed from *Fixed*(10) to *Normal* (10, 2.5). Moreover, the routing-out rule for W_{3a} is changed from *Circulate* to *Percent* (50%, 50%), which also introduces some stochastic property into the model. It is due to the fact that the destination is decided randomly based on the specified percentage going to each destination. As before, both a

standalone model and distributed simulation are created for the CSPE. Table 4.4.2 shows the experimental results for the stochastic model in all three cases. As with the deterministic model, CSPE(SA) and CSPE(DS) generate identical results since the same random seed is used in both versions of the stochastic model. The results between Simul8 and CSPE are also almost identical, showing the correctness of the CSPE. The minor differences between the CSPE and Simul8 are mainly due to different ways of generating random numbers. It can be seen that the percentage of the entities to be sent to different destinations by W_{3a} is nearly (50%, 50%) for all three cases. The experiments show that the CSPE can support both standalone and distributed simulations, and generate correct simulation results.

Table 4.4.2: Experiment results for stochastic model

		Simul8	CSPE(SA)	CSPE(DS)
En_{1a}	Arrival Entities	5000	5000	5000
En_2		5000	5000	5000
En_{1a}	Refused Entities	0	0	0
En_2		0	0	0
Q_{1a}	Total Entered Entities	5000	5000	5000
Q_{1b}		4976	4894	4894
Q_2		5000	5000	5000
Q_{3a}		9947	9869	9869
Q_{3b}		4920	4969	4969
Q_{3c}		4994	4994	4994
Q_{1a}	Queue Length at End Time	16	15	15
Q_{1b}		0	0	0
Q_2		0	0	0
Q_{3a}		47	1	1
Q_{3b}		0	0	0
Q_{3c}		2534	2509	2509
W_{1a}	Completed Entities	4983	4984	4984
W_{1b}		4975	4894	4894
W_2		4999	4999	4999
W_{3a}		9899	9867	9867
W_{3b}		2459	2484	2484
W_{1a}	Status at End Time	busy	busy	busy
W_{1b}		busy	idle	idle
W_2		busy	busy	busy
W_{3a}		busy	busy	busy
W_{3b}		busy	idle	idle
Ex_1	Completed Entities	2456	2481	2481

4.4.1.2 Performance Comparison between Standalone and Distributed Simulation

It is also interesting to compare the performance between the standalone and distributed simulations, which is one of the motivations for distributed simulation. The comparison is also conducted based on the same bicycle manufacturing system described in section 4.4.1.1.

The experiments are designed by varying two critical factors. These two factors play an important role for distributed simulation. One is the lookahead, which is critical for a distributed simulation adopting the conservative synchronization approach. Lookahead is used in the RTI to increase the degree of concurrency in the distributed simulation, i.e. when the DSManager requests to advance time on behalf of the simulation model, the time granted by the RTI is dependent on the simulation time and lookahead of other federates in the distributed simulation. Lookahead therefore contributes to the amount of progress in simulation time a CSP can make against real time. The other factor is event granularity, which is defined as the computation time taken to process an event. The bicycle manufacturing system represents a typical simulation model, but it is not complicated or large enough to demonstrate the benefits of the distributed simulation. For research purposes, event granularity is used to vary the computation time (real time) to reflect the actions taken during the execution of an event (e.g., updating of statistical counters, saves to a trace file) and therefore to compare the sharing of the processing demands of the simulation over multiple computers against a standalone simulation.

The experimental setup is same as that in section 4.4.1.1. The RTI used is the DMSO RTI1.3NG-V6. One computer is used to run the rtiexec, the initialization process for the RTI, and the other three for three separate federates (WPL, FPL and BAL models respectively). The *event granularity* is varied from 0.0001, 0.001 to 0.01 to investigate the effect of varied processing demands of the distributed simulation. For each event granularity the lookahead is 0, 20, 50 and 100. The lookahead is calculated based on the travel time between models. As shown in Figure 4.4.1, in the deterministic simulation, the travel time before the exit point is set as *fixed(100)*. For example, if an entity finishes processing in W_2 at time *t*, the earliest time it arrives at BAL is *t+100*. Thus, a lookahead between 0 and 100 is a safe value.

Table 4.4.3: Performance results for BMS distributed simulation (deterministic)

Event granularity (seconds)	Lookahead	Execution time (seconds)		Speedup
		Distributed Simulation	Standalone Simulation	
0.0001	0	25.1194	3.1714	0.126253
	20	13.0584		0.2428628
	50	12.041		0.2633834
	100	12.4631		0.2544632
0.001	0	40.2601	19.5934	0.4866704
	20	23.2505		0.8427088
	50	22.9684		0.853059
	100	22.5420		0.8691953
0.01	0	194.5795	183.9713	0.9454814
	20	110.2301		1.6689752
	50	98.7643		1.8627308
	100	96.5369		1.9057096

Table 4.4.3 shows the results from the experimentation. The simulations are run for 100,000 time units with each experiment run three times to derive an average. The execution time for the distributed simulation is for the last federate to reach the end time. Here speedup is used to compare the performance between standalone and distributed simulations. Using the definition adopted in parallel computing [QUI03], speedup is calculated as the ratio between the sequential execution time and the parallel execution time:

$$\text{Speedup} = \frac{\text{Sequential execution time}}{\text{Parallel execution time}}$$

Similarly, speedup is defined here as the ratio between the standalone execution time and the distributed execution time:

$$\text{Speedup} = \frac{\text{Standalone execution time}}{\text{Distributed execution time}}$$

Figure 4.4.3 shows speedup vs. lookahead with varying event granularities. As can be seen, performance for this distributed simulation improves as both event granularity and lookahead increases.

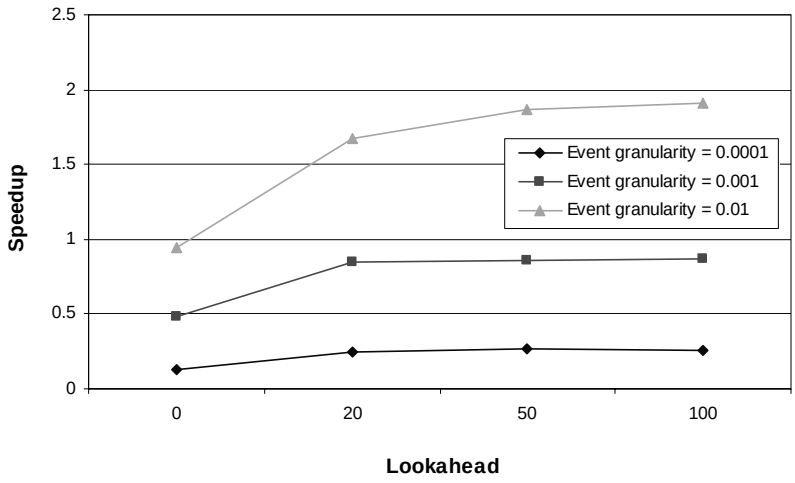


Figure 4.4.3: Speedup vs. lookahead for BMS distributed simulation

The results of the experiments show that as event granularity increases, so does the performance of the distributed simulation as compared to the standalone one. It is due to the fact the computation load can be processed concurrently. Additionally, the performance is also improved with the increase of the lookahead. These results indicate for this Type I IRM problem it is possible to achieve some kind of performance benefit, one of the motivations of distributed simulation. The performance benefit is expected to improve with more complex models consisting of, for example, many workstations and routings.

In the above experiments, lookahead is exploited using a fixed travel time. Another set of experiments is conducted using stochastic travel time for event granularity of 0.01 seconds. The results are presented in Table 4.4.4. As can be seen the lookahead values of 0, 20, 50, 65, 80 and 100 are used. Values of 0, 20, 80 and 100 represent the lower bound on travel time that a simulation modeler might place on this distribution. It is achieved by refusing the sampled values which are lower than the bounded value. For example, if a value of 63 is calculated using *normal(100, 10)* and the lookahead of 80 is applied, 63 is refused and new values are calculated till a value is generated larger than 80. Statistically, this may be inappropriate for general simulation models but this experiment is concerned is simply with performance. It should be noted that the CSP can also calculate the value of the lookahead automatically by presampling the possible values based on the random stream. Specifically, 65 represents the lowest possible value that can be sampled from the normal distribution *normal(100, 10)* based on the random number stream.

Table 4.4.4: Performance results for BMS distributed simulation (stochastic)

Event granularity (seconds)	Travel time Normal(100,10)	Lookahead	Execution time (seconds)		Speedup
			Distributed Simulation	Standalone Simulation	
0.01	100 (min:65)	0	231.9937	183.8887	0.7926452
		20	114.9736		1.5993993
		50	99.4486		1.8490828
		65	97.8397		1.8794896
	100 (min:80)	80	97.3224	183.8811	1.8894016
	100 (min:100)	100	97.084	183.8951	1.8941854

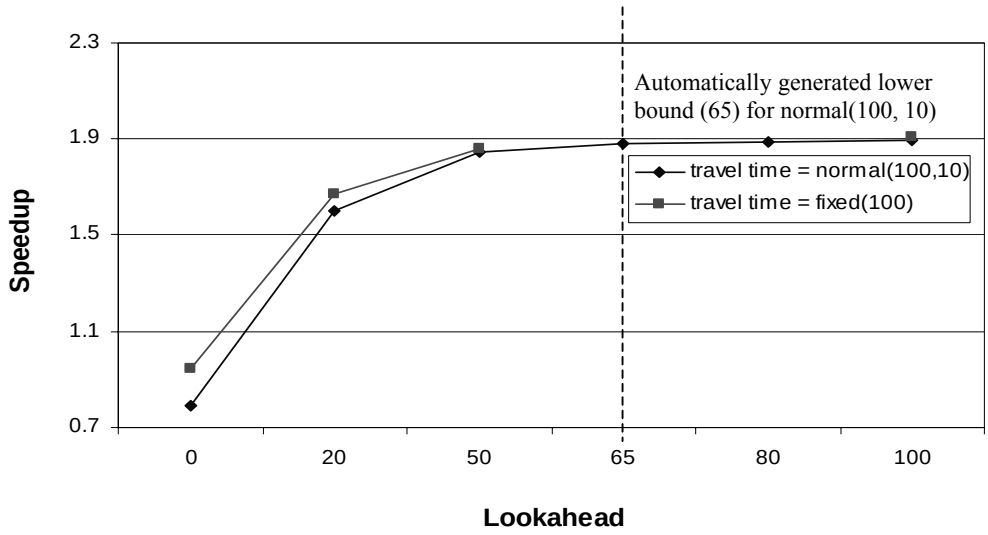


Figure 4.4.4: Speedup vs. lookahead for BMS (event granularity = 0.01) using deterministic / stochastic travel time

Figure 4.4.4 shows the resulting speedup compared with that using fixed travel time. The slight drop in performance can be explained by the variance in the events generated from the normal distribution (with possibly fewer events to process per time advance request). It also shows that the performance increases with larger lookahead value, demonstrating the crucial role for lookahead in distributed simulation using conservative synchronization approaches. It is possible to exploit a lookahead value in the Type I IRM. For other types of IRMs, some problems may exist to calculate a large lookahead, which should be carefully considered.

4.4.2 CSPI-PDG Type II IRM

Another set of experiments is designed to test the proposed solutions for Type II IRM synchronous entity passing. The experiments are conducted using the CSPE which is linked with the DSManager for the Type II IRM. First, to evaluate the simulation results are correct, Simul8 is chosen to run a standalone simulation for the same simulation model.

4.4.2.1 Evaluation of the Correctness of the CSPE and Interface

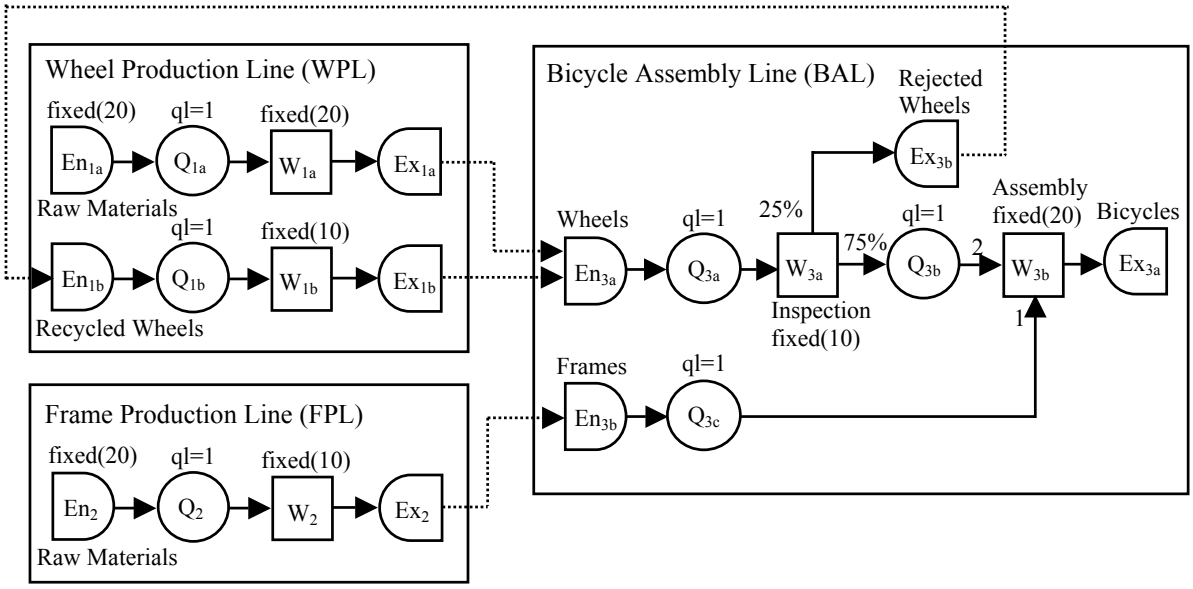


Figure 4.4.5: The Type II bicycle manufacturing system (distributed & deterministic)

Figure 4.4-5 shows a distributed and deterministic simulation for the bicycle manufacturing system similar to that in section 4.4.1. The major difference is the bounded queues in each model which are introduced to demonstrate the Type II IRM. The maximum length of all the queues (ql) is set as 1, 10 and 100 respectivly. A corresponding standalone and deterministic model is shown in Figure 4.4.6.

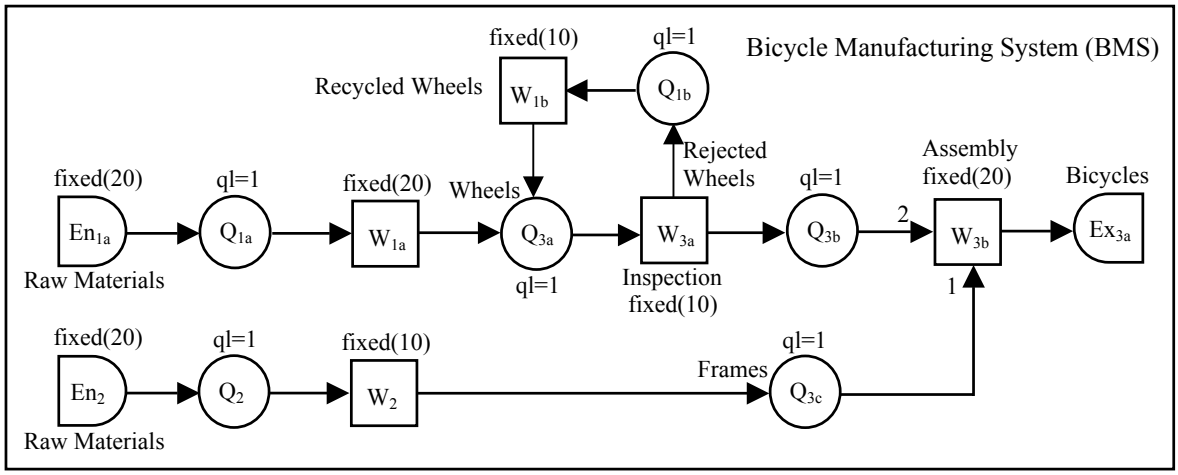


Figure 4.4.6: The Type II bicycle manufacturing system (standalone & deterministic)

Moreover, another set of experiments is carried out for stochastic models by introducing some probability distributions into the system. Instead of a fixed distribution, a normal distribution is used for the processing time in all workstations. For instance, the processing time of W_{1a} is changed from *Fixed* (20) to *Normal* (20, 5) and the routing-out rule for W_{3a} is changed from *Circulate* to *Percent* (25%, 75%), which also introduces some stochastic property into the model. It is due to the fact that the destination is decided randomly based on the specified percentage going to each.

Table 4.4.5 shows the experimental results for simulating the system for 100,000 time units in Simul8, CSPE(SA) (standalone model) and CSPE(DS) (distributed simulation) with a maximum queue length of 1. The final throughput of the system as well as the statistics for each simulation object are identical for all three cases when the deterministic model is used, showing the correctness of the CSPE and successful interoperability of Type II IRMs. As for the stochastic model, CSPE(SA) and CSPE(DS) generate identical results. The results between Simul8 and CSPE are also almost identical, showing the correctness of the CSPE. The minor differences between the CSPE and Simul8 are mainly due to different ways of generating random numbers. With a queue length of 10 and 100, similar experimental results are generated (see Table 4.4.6 and Table 4.4.7). These results show that the CSPE integrated with the DSManager for the Type II IRM can generate correct simulation statistics, indicating the status information is successfully transferred between the models.

Table 4.4.5: Experimental results for distributed & standalone simulation on CSPE and Simul8 (ql=1)

		Deterministic			Stochastic		
		Simul8	**CSPE(SA)**	**CSPE(DS)**	**Simul8**	**CSPE(SA)**	**CSPE(DS)**
En_{1a}	Arrival Entities	5000	5000	5000	5000	5000	5000
En_2		5000	5000	5000	5000	5000	5000
En_{1a}	Refused Entities	4496	4496	4496	4843	4822	4822
En_2		4747	4747	4747	4921	4910	4910
Q_{1a}	Total Entered Entities	504	504	504	157	178	178
Q_{1b}		500	500	500	47	64	64

Q_2		253	253	253		79	90	90
Q_{3a}		1001	1001	1001		200	238	238
Q_{3b}		499	499	499		151	172	172
Q_{3c}		251	251	251		77	88	88
Q_{1a}	Queue Length at End Time	1	1	1		1	1	1
Q_{1b}		0	0	0		1	1	1
Q_2		1	1	1		1	1	1
Q_{3a}		1	1	1		1	1	1
Q_{3b}		0	0	0		0	0	0
Q_{3c}		1	1	1		1	1	1
W_{1a}	Completed Entities	503	503	503		156	177	177
W_{1b}		500	500	500		46	63	63
W_2		252	252	252		78	89	89
W_{3a}		999	999	999		199	237	237
W_{3b}		249	249	249		75	86	86
W_{1a}	Status at End Time	busy	busy	busy		busy	busy	busy
W_{1b}		busy	busy	busy		busy	busy	busy
W_2		busy	busy	busy		busy	busy	busy
W_{3a}		busy	busy	busy		busy	busy	busy
W_{3b}		busy	busy	busy		busy	busy	busy
Ex_1	Completed Entities	249	249	249		75	86	86

Table 4.4.6: Experimental results for distributed & standalone simulation on CSPE and Simul8 (ql=10)

		Deterministic				**Stochastic**		
		Simul8	**CSPE(SA)**	**CSPE(DS)**		**Simul8**	**CSPE(SA)**	**CSPE(DS)**
En_{1a}	Arrival Entities	5000	5000	5000		5000	5000	5000
En_2		5000	5000	5000		5000	5000	5000
En_{1a}	Refused Entities	4478	4478	4478		4217	4238	4238
En_2		4729	4729	4729		4598	4608	4608
Q_{1a}	Total Entered Entities	522	522	522		783	762	762
Q_{1b}		500	500	500		238	259	259
Q_2		271	271	271		402	392	392
Q_{3a}		1010	1010	1010		1010	1010	1010
Q_{3b}		499	499	499		761	740	740
Q_{3c}		260	260	260		391	381	381
Q_{1a}	Queue Length at End Time	10	10	10		10	10	10
Q_{1b}		0	0	0		0	0	0
Q_2		10	10	10		10	10	10
Q_{3a}		10	10	10		10	10	10
Q_{3b}		0	0	0		0	0	0
Q_{3c}		10	10	10		10	10	10
W_{1a}	Completed Entities	512	512	512		773	752	752
W_{1b}		500	500	500		238	259	259
W_2		261	261	261		392	382	382
W_{3a}		999	999	999		999	999	999
W_{3b}		249	249	249		380	370	370
W_{1a}	Status at End Time	busy	busy	busy		busy	busy	busy
W_{1b}		busy	busy	busy		idle	idle	idle
W_2		busy	busy	busy		busy	busy	busy
W_{3a}		busy	busy	busy		busy	busy	busy
W_{3b}		busy	idle	idle		busy	idle	idle
Ex_1	Completed Entities	249	249	249		380	370	370

Table 4.4.7: Experimental results for distributed & standalone simulation on CSPE and Simul8 (ql=100)

		Deterministic				Stochastic		
		Simul8	**CSPE(SA)**	**CSPE(DS)**		**Simul8**	**CSPE(SA)**	**CSPE(DS)**
En_{1a}	Arrival Entities	5000	5000	5000		5000	5000	5000
En_2		5000	5000	5000		5000	5000	5000
En_{1a}	Refused Entities	4298	4298	4298		4037	4058	4058
En_2		4549	4549	4549		4418	4428	4428
Q_{1a}	Total Entered Entities	702	702	702		963	942	942
Q_{1b}		500	500	500		238	259	259
Q_2		451	451	451		582	572	572
Q_{3a}		1100	1100	1100		1100	1100	1100
Q_{3b}		499	499	499		761	740	740
Q_{3c}		350	350	350		481	471	471
Q_{1a}	Queue Length at End Time	100	100	100		100	100	100
Q_{1b}		0	0	0		0	0	0
Q_2		100	100	100		100	100	100
Q_{3a}		100	100	100		100	100	100
Q_{3b}		0	0	0		0	0	0
Q_{3c}		100	100	100		100	100	100
W_{1a}	Completed Entities	602	602	602		863	842	842
W_{1b}		500	500	500		238	259	259
W_2		351	351	351		482	472	472
W_{3a}		999	999	999		999	999	999
W_{3b}		249	249	249		380	370	370
W_{1a}	Status at End Time	busy	busy	busy		busy	busy	busy
W_{1b}		busy	busy	busy		idle	idle	idle
W_2		busy	busy	busy		busy	busy	busy
W_{3a}		busy	busy	busy		busy	busy	busy
W_{3b}		idle	idle	idle		busy	idle	idle
Ex_1	Completed Entities	249	249	249		380	370	370

4.4.2.2 Performance Comparison between Standalone and Distributed Simulation

A performance comparison is also made between the standalone and distributed simulation for deterministic and stochastic distributed simulations respectively. As before the speedup is also calculated as the ratio between the standalone execution time and the distributed execution time. As can be seen in Figure 4.4.7 and Figure 4.4.8, the speedup is less than 1 in both cases, which indicates the standalone simulation outperforms the distributed simulation. It is because of the introduction of near zero lookahead in the models. In addition, the results also show that the performance of the distributed simulation is approaching that of the standalone simulation with increased event granularity. This is because the communication overhead introduced by distributed simulation is not so obvious when the computation load is large enough. In the above, it can be seen that the lookahead is difficult to exploit in the Type II IRM due to the requirement of timely updating of status information. It is worthwhile to see how optimistic synchronization approaches can achieve

any benefit in this type of IRM. In Chapter 5, an approach to address performance effectiveness for the case of Type II IRM using optimistic synchronization will be described.

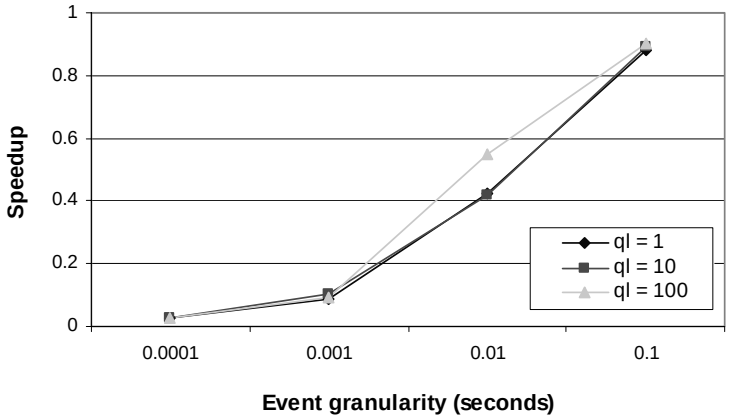


Figure 4.4.7: Experimental results for deterministic and distributed models with different queue length

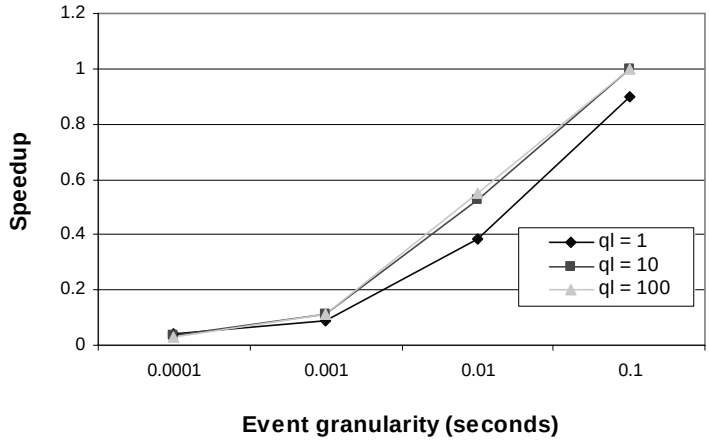


Figure 4.4.8: Experimental results for stochastic and distributed models with different queue length

It is also interesting to investigate the overhead introduced by the new features in the DSManager in situations where the EEP is not restricted. Another set of experiments is carried out using a Type I IRM, the same BMS (deterministic model) except all the queues are unbounded. The experimental results are compared between the CSPE with DSManager for Type I IRM and the CSPE with DSManager for Type II IRM. It is observed the simulation results are identical and only around 2 more seconds are spent in execution time using the Type II DSManager, 36.56 seconds as compared to 34.38 seconds using the Type I DSManager. It is not a large overhead and optimization will be applied to the DSManager in the future.

4.4.2.3 External Entry Point with Multiple Priorities

To test the correctness of the CSPE and interface for the Type II IRM with EEPs having multiple priorities, another distributed simulation is created, consisting of 4 models M_1, M_2, M_3 and M_4. M_1, M_2 and M_3 transfer entities to two workstations with fixed capacity in M_4 via three EEPs. As shown

in Figure 4.4.9, En_{4b} has lower priority than En_{4a} for W_{4a}, but has higher priority than En_{4c} for W_{4b}. So it is possible the priority of En_{4b} may be changed dynamically when an entity is passed to a different workstation.

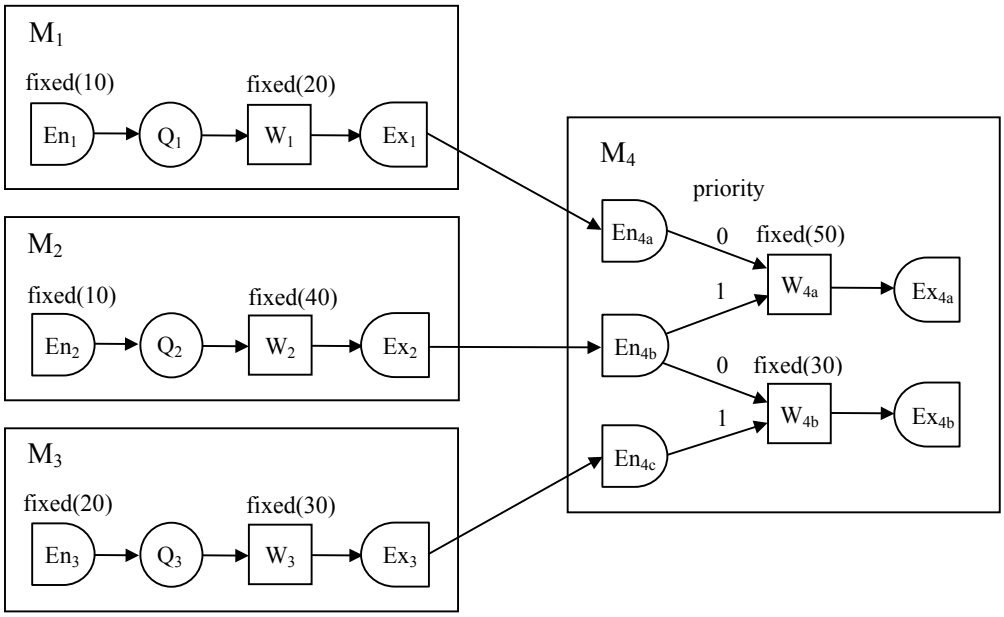


Figure 4.4.9: Type II IRM with external entry points having multiple priorities

Table 4.4.8: Experimental Results for Type II IRM with external entry points having multiple priorities

		Simul8	CSPE(SA)	CSPE(DS)
En_1	Arrival Entities	100	100	100
En_2		100	100	100
En_3		50	50	50
En_1	Refused Entities	0	0	0
En_2		0	0	0
En_3		0	0	0
Q_1	Total Entered Entities	100	100	100
Q_2		100	100	100
Q_3		50	50	50
Q_1	Queue Length at End Time	79	79	79
Q_2		83	83	83
Q_3		33	33	33
W_1	Completed Entities	21	21	21
W_2		17	17	17
W_3		16	16	16
W_{4a}		19	19	19
W_{4b}		31	31	31
W_1	Status at End Time	busy	busy	busy
W_2		busy	busy	busy
W_3		busy	busy	busy
W_{4a}		busy	busy	busy
W_{4b}		busy	busy	busy
Ex_{4a}	Completed Entities	19	19	19
Ex_{4b}		31	31	31

Table 4.4.8 shows the experimental results for simulating the system for 1,000 time units in Simul8, CSPE(SA) and CSPE(DS). It can be seen the distributed simulation produces identical results to the standalone simulation using CSPE or Simul8. This proves that priority as well as status information is correctly transferred between different models. Also the inter-model simultaneous events are processed in the correct order when updating the priority dynamically. The above experiments show that the CSPE integrated with the new DSManager can also support those special models with EEPs having multiple priorities.

4.5 Summary

Based on the survey of some popular CSPs, the CSP Emulator (CSPE) is developed to investigate the requirements for the integration of CSPs with the HLA. Its simulation engine adopts the three phase approach to manage the activities of the entities and trace the life history of each entity in a simulation. In addition to supporting standalone simulation in the same way as current CSPs, the CSPE also provides the functionality for the creation of a model component to join a distributed simulation. It is achieved by developing special features in the CSPE, including the interface used by the modeler and the internal simulation engine. The CSPE should offer a friendly user interface to allow the modeler to declare the interoperability information. Internally, the CSPE needs to communicate with the DSManager and forward the interoperability information to it as well as to process the information received from the DSManager. Currently, the CSPE can be linked with the DSManager designed for CSPI-PDG Type I IRM and Type II IRM.

Since the features of the CSPE are developed based on the survey of some popular CSPs, the correctness of the CSPE as well as the generic interface is validated by comparing the simulation results for a typical CSPI-PDG Type I IRM (both deterministic and stochastic models) between CSPE and Simul8, one of the popluar CSPs. In addition, experiments are also carried out for the Type I IRM to compare the performance between standalone and distributed simulation by varying two factors: event granularity and lookahead. The experiments show that for simulations with large computation overhead, the distributed simulation can achieve better performance with the exploitation of larger lookahead. Based on the CSPE, the proposed solutions for the Type II IRM are also verified by comparing the simulation results between the CSPE and Simul8. The CSPE integrated with the Type II DSManager generates identical results to Simul8 for the deterministic model, and almost identical for the stochastic model, showing the correctness of the DSManager for the Type II IRM synchronous entity passing problem. However, the performance of the distributed simulation is worse than that of the standalone simulation in the experiments. It is because of the existence of near zero lookahead introduced by the need to timely update the status information in

the Type II IRM. Furthermore, the new DSManager designed for the Type II IRM using the modified DMSO RTI1.3NG-V6 logical time can also be applied for the Type I IRM without too much overhead. In this way, the model only needs to inform the DSManager whether each local EEP is restricted (linked with a bounded queue or a workstation with limited capacity) or not, without identifying the type of the model itself.

In summary, using the CSPE, the requirements of different types of IRMs can be investigated and alternative interoperability solutions can be benchmarked. The CSPE also provides a suggestion as to how current CSPs may be modified to provide HLA capability to support distributed simulation. Based on the work of the integration of the CSPE with the HLA, the generic interface has also been applied for two actual CSPs, Autosched AP and IBM's WBI Modeler, to support the interoperability of distributed simulation models. This will be described in more detail in Chapter 7.

CHAPTER

5

OPTIMISTIC SIMULATION WITH THE HLA

5.1 Introduction

Current CSP-HLA interfaces usually adopt the conservative synchronization approach because of the complexity of handling the rollback procedure within the modelers' simulations. However, the main constraint for the conservative approach is the need for lookahead to achieve good performance. Lookahead is a guarantee from a federate that it will not generate any external message with a timestamp smaller than its current time plus the value of lookahead. If the guarantee is zero, then the conservative protocols may perform poorly. Therefore, it is especially interesting to see whether established optimistic synchronization protocols can be integrated into existing simulation systems in a manner which does not require major user involvement [STR01].

In this chapter, a rollback controller is introduced that uses a middleware approach to handle the rollback procedure on behalf of the simulation model. The rollback controller provides communication with the HLA in a way that hides details of the HLA from the modelers, which facilitates reuse and interoperability of simulation models using CSPs. To solve the problem of an error in the 1.3 HLA interface specification, a new time advance algorithm is proposed that can fully utilize the benefits of optimistic synchronization. A comparison of performance between the conservative and optimistic synchronization approaches is also provided using a typical CSPI-PDG Type II IRM, a pipe line factory with bounded queues in the models. Additionally, this chapter also describes a scalability study showing the experimental results for the two synchronization approaches as the number of simulation components increases.

5.2 HLA-based Optimistic Simulation

Little work has been done on HLA-based optimistic simulation. Even though the HLA offers some services for the optimistic approach, the rollback mechanisms provided by the RTI cover only event retractions. All the management of state saving and recovery is still required to build optimistic federates. In [FPF00], Ferenci et al. describe an implementation of a federation of optimistic simulators. Using libraries of their Federation Development Kit (FDK) [FDK], an RTI was built that implements a subset of the HLA services. Specifically, each federate is an instance of the GTW (Georgia Tech Time Warp) optimistic parallel simulator. However, their work does not attempt to address the general problem of interoperability and reuse of arbitrary legacy simulators since only the GTW simulator can be supported in the federation.

To relieve the modeler from the burden of handling problems in the optimistic approach, a rollback manager was advocated by Vardânega and Maziero in [VAM99] [VAM00] to deal with the management of state saving and recovery. A problem with their rollback manager is that it introduces some conservative elements into the optimistic approach when the federate requests time advance. It cannot fully exploit the advantages of the optimistic approach but still has the optimistic overhead. In addition, no simulation model was selected to compare the performance of the optimistic and conservative approaches, which would allow the investigation of the most suitable synchronization approach for different simulation scenarios.

Compared to the conservative approach, there are more complications for introducing the optimistic approach into CSP-HLA interfaces. One complication is that the CSPs must offer a technique for state saving and restoring the system state for recovering from possible causality errors. However, some CSPs have provided such functionality. GPSS/H [HEC95] offers the CHECKPOINT statement and Simplex3 [SCH00] allows the user to schedule breaks within model experiments, both of which save the entire model state to hard disk. Some other CSPs, e.g., Witness [MAM97], have a replay service which also records the model state at runtime. Though the performance overhead of the state saving operations in these approaches is high, it is worth considering the optimistic approach since these techniques present the possibility of extensibility towards optimistic synchronization. The other complication is the burden of handling problems related to the rollback management. Similar to the work done by Vardânega and Maziero, a rollback controller is introduced that uses a middleware approach to realize the detailed implementation of the rollback procedure. However, a time advance algorithm is proposed in this thesis that can fully exploit the benefits of the optimistic approach.

5.3 HLA Facilities for Optimistic Simulation

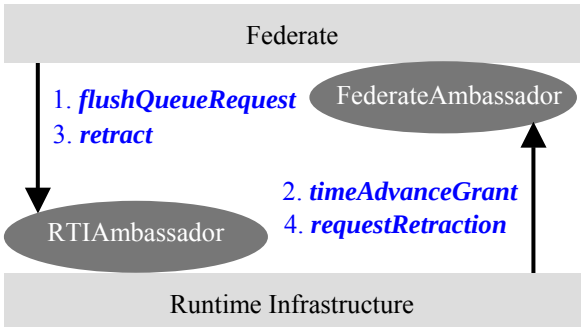


Figure 5.3.1: Time advance and retraction in the HLA

In the HLA, both conservative and optimistic synchronization approaches are supported. In the optimistic approach, all events can be delivered to the federate without considering causality. So it is possible that another event containing a smaller timestamp will arrive later, possibly from a different federate. When such an event arrives, some incorrect computations must be rolled back. Performing rollback means restoring the current state to a previous "safe" state that has been recorded and canceling all the events scheduled by the incorrect computations. If the cancelled events have already been processed at other federates, those federates also need to roll back. Recursively applying this procedure will eventually erase the effects of the incorrect computation [JEF85]. The HLA supports an optimistic federate by providing the following services to realize optimistic execution.

- *flushQueueRequest*(). Optimistic federates must be able to receive TSO (Timestamp Order) messages before the RTI can guarantee that no smaller timestamped message will later be received in order to allow optimistic event processing. The *flushQueueRequest*() service forces the RTI to deliver all buffered events in the internal queue of the local RTI component.

- *retract*(). To ensure that optimistic federates are able to cancel previous incorrect events which have been sent out, the *retract*() service is provided to withdraw an update, interaction, or deletion previously scheduled by the federate. If the specified event is currently queued for delivery to a given remote federate, it is removed from its queue. If this event has been delivered to the remote federate, the appropriate callback *requestRetraction*() is invoked and that federate is responsible for rolling back its state as appropriate (see Figure 5.3.1).

- Another important issue for optimistic federates is fossil collection. An optimistic federate must be able to compute a lower bound, called Global Virtual Time (GVT), on the logical

time of any future rollback. Irrevocable operations occurring at a simulation time older than GVT can be performed, and storage carrying a timestamp older than GVT can be reclaimed. In the HLA the time granted by the RTI after a request to advance time can be used as the GVT value.

5.4 Implementation of Rollback Controller Using a Middleware Approach

To relieve the simulation modeler from the burden of handling the complex details about the rollback procedure, a middleware approach is used to extend the original services in the RTI APIs. All the services the federate calls will be intercepted by a rollback controller in the middleware, which implements the rollback procedure on behalf of the federate. The rollback controller is responsible for saving the states, restoring a "safe" state, retracting incorrectly scheduled events and handling the retractions received from other federates. It is important that all these processes are done as transparently as possible to the simulation modelers. They can still build their simulation federate as before, but instead of linking the RTI library to their simulation programs, the middleware library RTI+ is linked (as discussed in Section 3.2). Figure 5.4.1 shows the extended RTI architecture with the rollback controller.

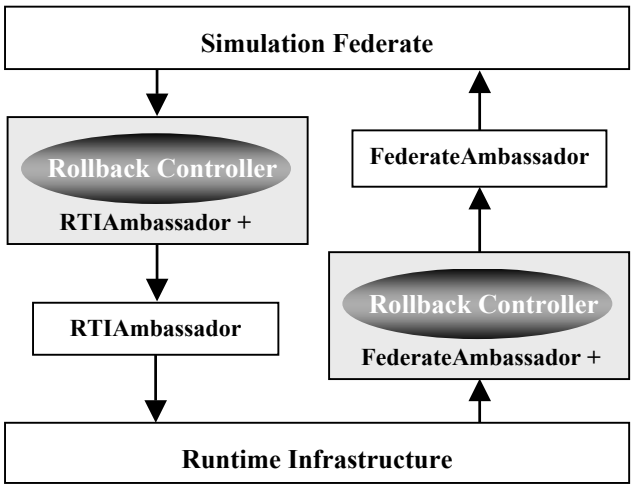


Figure 5.4.1: Rollback controller in middleware

The RTI+ provides all the RTI ambassador interfaces, and filters the federate ambassador callbacks through the FederateAmbassador+ before passing them to the user's FederateAmbassador. The federate will continue to call the same methods of the RTIAmbassador class to interact with the RTI and it will receive RTI callbacks through the same FederateAmbassador class methods. However, some time management method calls related to the rollback procedure will be intercepted by the

RTIAmbassador+ and FederateAmbassador+, and then passed to the rollback controller. All the other methods are passed directly to the RTIAmbassador and FederateAmbassador. Using this approach, the federate can adopt an optimistic behavior without worrying about the handling of rollbacks.

5.4.1 The Structure of the Rollback Controller

Table 5.4.1: Abbreviation of RTI services

RTIAmbassador		**FederateAmbassador**	
FQR	*flushQueueRequest*	TAG	*timeAdvanceGrant*
NER	*nextEventRequest*	RAV	*reflectAttributeValues*
NERA	*nextEventRequestAvailable*	RI	*receiveInteraction*
UAV	*updateAttributeValues*	ROI	*removeObjectInstance*
SI	*sendInteraction*	RR	*requestRetraction*
DOI	*deleteObjectInstance*		

In the following description, the RTI services are abbreviated as shown in Table 5.4.1. The structure of the rollback controller is first described based on the optimistic time management introduced in [JEF85]. Since the rollback controller is designed for the integration of CSPs, the CSP's internal event queue (used to store the scheduled events) and external event queue (used to store the events received by the FederateAmbassador) cannot be accessed by the controller and should be saved as part of the federate's state. However, the controller will keep track of the events recently forwarded to the FederateAmbassador and when these events are forwarded. Therefore, the rollback controller class has the following attributes:

- A local virtual clock indicates the federate's current logical time t_c. The current time should be defined as the timestamp of the event being processed. To achieve this, a new service named "*processEvent*(t)" is provided in RTIAmbassador+, where t is the timestamp of the event to be processed.

- The rollback time t_{rb} stores the timestamp of the event which causes a local causality violation. A causality violation occurs when an event with timestamp less than t_c is received, or a previously processed event is retracted.

- A *FederateState* queue contains saved copies of the federate's recent states. In order to make rollback possible, the rollback controller must save the state of the federate from time to time. The state includes both the internal event queue and the external event queue. Each

state in the *FederateState* queue is associated with a corresponding time t_s which is the current t_c when the state is saved.

- An *InEvent* queue contains all recently arrived messages sorted in order of their timestamps. Some of these messages have already been processed because no local causality error was detected at that time. Nevertheless, they are not discarded from the queue because it may be necessary to roll back and reprocess them. Each event in this queue is associated with a processed time t_p indicating when the event is forwarded to the federate.

- A *nextEventIndicator* points to the next event in the *InEvent* queue to be forwarded to the federate. Instead of forwarding all the received events from FQR to the federate, the rollback controller only forwards those events with the smallest timestamp. This can throttle the optimism and avoid the federate rolling back too many steps when a RR callback is received.

- An *OutEvent* queue contains copies of the messages the federate has recently sent, also kept in timestamp order. The rollback controller keeps track of all messages sent and their retraction handles. These messages are needed in case of rollback, in order to "unsend" them to other remote federates. These outgoing messages act as the anti-messages in the optimistic mechanism. Each event in the *OutEvent* queue is associated with a scheduled time t_s indicating when the event is sent out to remote federates. When the rollback occurs, all those events in the *OutEvent* queue with scheduled time $t_s > t_{rb}$ should be retracted.

In addition to these attributes, the rollback controller class has some corresponding member functions to manage the *FederateState* queue, the *InEvent* queue and the *OutEvent* queue. Only the states or events with timestamp greater than or equal to GVT need to be retained.

5.4.2 State Checkpoint Mechanism

In order to restore some previous state when a rollback occurs, the rollback controller should keep periodic snapshots of the federate's internal state (state checkpoints). This requires the rollback controller to access the federate's state at any time. To provide that functionality, each federate should implement two callback methods that give controlled access to its internal state, to allow the rollback controller to save the current state or to restore a previous state: (1) *getState*(S_c), which returns the federate's current state S_c which is recorded at the current time t_c, (2) *setState*(S_p), which restores the federate's state to a previous state S_p. The rollback controller uses the *getState* method to add a new federate state into the *FederateState* queue and the *setState* method to restore a previous state from the *FederateState* queue, when a rollback occurs. Since a transparent approach is adopted,

the user federate need not consider the management of this state queue. What it should do is only record its current state and set the "safe" state from the rollback controller. An important issue to be addressed is when the rollback controller should call *getState*(S_c) or *setState*(S_p).

When the rollback controller invokes *getState*(S_c), it means a checkpoint should be recorded. The new state associated with the current time t_c is combined as a new record, which will be inserted into the state queue. The rollback controller currently uses the copy state-saving mechanism which means the whole state is recorded prior to the execution of each event. This can be realized when the federate calls the new service *processEvent*(t) in the RTIAmbassador+. In addition, the rollback controller also saves the state before forwarding the received events to the FederateAmbassador. Though forwarding the events will not change t_c, these states are saved to avoid the federate rolling back more steps than needed. When a *RR* is received, the federate only rolls back to the state before the event is forwarded, not a previous state recorded before processing events by the federate. To reduce the memory overhead, only the first state among those with the same t_c is saved. In the HLA, there are three major event types with timestamps that may be received by the federate: *RAV*, *RI* and *ROI*. These events should be managed separately by the rollback controller, to allow it to maintain control on all modifications performed on the federate state.

The rollback controller invokes *setState*(S_p) in the situation that a local causality error has occurred and the rollback operation must be invoked. A causality error can only be detected in TSO events *RAV*, *RI* and *ROI* or a *RR* event issued from other federates. In this case, the rollback controller searches through the Federate State queue to look for a state S_l with associated timestamp t_l satisfying this condition: $t_l \leq t_{rb} < t_u$, where t_l and t_u are the adjacent timestamps of the Federate states. Then it calls *setState*(S_l) to restore the federate's state to a previous "safe" state. However, this "safe" state may not be really safe since another message older than t_l may possibly be received later and a state earlier than S_l must be restored.

The rollback controller still requires the model to provide the state to perform checkpointing/recovery. Subsequent to the research described in this thesis, some work has been done to further release the modeler from such a burden. In [SAQ05a] [SAQ05b], a MASM (Magic State Manager) is employed to perform the checkpointing by tracking modified memory pages using the special features of any Posix compliant operating system (e.g., the Linux operating system), which realizes an approach completely transparent to the model.

5.4.3 Local Causality Violation Detection and Rollback Procedure

An optimistic federate will typically process messages containing a timestamp larger than GVT, giving rise to optimistic execution. Any number of messages may be processed before the RTI queue is again flushed. Such optimistic execution may incur a local causality violation and then trigger the rollback procedure. These two processes are described as follows:

5.4.3.1 Local Causality Violation Detection

As previously discussed, a causality error may be detected in the three types of TSO messages: $RAV(t_r)$, $RI(t_r)$ or $ROI(t_r)$. When receiving such a TSO message the controller will first invoke $getState(S_c)$ to record the federate's old state, and then compare the timestamp t_r with the current logical time t_c (the rollback controller is at the same simulation time as the federate it manages). If $t_r \geq t_c$, the message is considered as a safe message and stored in *InEvent* Queue. Otherwise, a local causality violation is detected. Besides these TSO messages, another message *RR* may trigger a rollback procedure. The rollback manger will search its *InEvent* Queue to find a received event with this event handle. If such an event is found but has not yet been processed, it will not cause a rollback and only an annihilation is needed. Otherwise, if it has been processed, the rollback procedure will be started. However, there is a possibility the anti-message (via a *RR*) arrives before the corresponding positive message. In this case, the information in the *RR* method will be stored and the corresponding positive message will be annihilated when it is received.

5.4.3.2 Rollback Procedure

The rollback procedure is triggered by the causality error. It includes the reset of simulation time, restoring a safe state and canceling all incorrectly scheduled events to other federates. This procedure is handled by the rollback controller on behalf of the simulation model. Figure 5.4.2 shows a typical optimistic federate simulation with the rollback controller and the interactions between the federate, rollback controller and RTI. When the rollback controller detects that a causality violation occurs it interacts with the federate and RTI to execute the rollback procedure.

When the rollback procedure is triggered, the rollback time t_{rb} is set to the timestamp of the event causing the local causality error. The rollback controller should first "unsend" all the incorrect messages sent to other federates to reduce the effects of incorrect computations. So it searches its *OutEvent* queue and invokes the *retract*() method on the RTI to cancel all the incorrect events. The rollback controller stores all the received events in *InEvent* queue in timestamp order till a timeAdvanceGrant callback is made by the RTI. Before forwarding the grant time t_g to the federate,

the rollback controller will: (1) search the Federate State queue and find the state S_l with timestamp $t_l \leq t_{rb} < t_u$, where t_l and t_u are the adjacent timestamps of the Federate states, invoke *setState*(S_l) to restore the federate to a previous "safe" state and set the time associated with S_l as current time t_c; (2) update the value of *nextEventIndicator* and forward the events with the timestamp equal to t_{rb} in the *InEvent* queue to the federate. The *InEvent* queue contains the events received not only in the current flush but also in previous flushes (shown as $(RAV, RI, RO)_{old}$ in Figure 5.4.2); (3) discard all the events in *OutEvent* queue and the states in *FederateState* queue with timestamp larger than t_{rb} because they will not be needed for a future rollback procedure.

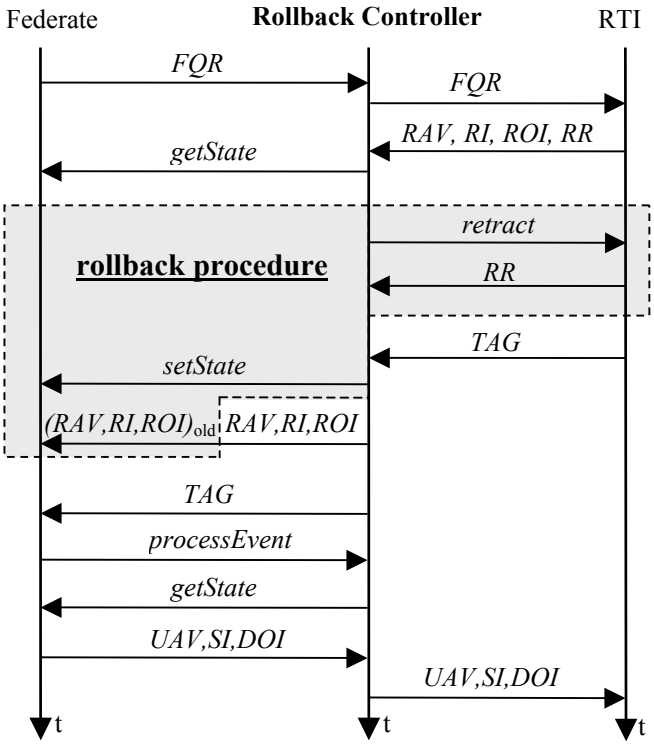


Figure 5.4.2: A typical optimistic federate simulation with rollback controller

5.5 Problem in the 1.3 HLA Interface Specification and Solution

Optimistic federates must be able to compute the GVT as a lower bound on the logical time of any future roll back. GVT allows the optimistic federate to reclaim memory resources, and to perform operations such as I/O that cannot be rolled back. In the HLA, since the time t_g granted by the RTI after the federate issues a request to advance time is a guarantee that all the TSO messages with timestamps smaller than the grant time have been delivered, it can be used as the GVT value to process fossil collection [FUJ98]. Optimistic federates use the *FQR* service to receive the events, and

then optimistically process those events. The time specified in this service provides local information for the RTI to compute t_g. Generally, the federate only needs to provide its next local event time as the specified cutoff time. However, an error in the 1.3 HLA interface specification is observed in the experiments.

According to the 1.3 HLA specification [RTI02], the grant time t_g for *FQR*(t) is the minimum of the minimum-next-event time and the specified cutoff time t. This grant time is used as a guarantee that no event with timestamp less than or equal to t_g will be received by the federate. However, it is not correct for an optimistic federate. The reason is that in *FQR* the minimum-next-event time is calculated based on the smallest timestamp of all TSO events that may be delivered to the federate in the future, without taking into account those which have been presented to the federate as a result of the flush. Thus, the federate may have received a TSO event and may then generate a new TSO event both with timestamp less than the grant time. This will incur an error indicating the timestamp is invalid and the event can not be successfully sent out.

The error is further explained by comparing the *NER* and *FQR* with an example. When the federate requests time advance using *NER* or *NERA*, the grant time is equal to the timestamp of the event(s) delivered to the federate or the specified cutoff time. Suppose there are two federates, A and B, as shown in Table 5.5.1, where A invokes *NER* (100) and B invokes *NER* (200). Suppose there is an event with timestamp 50 received by A, the time granted by the RTI will be 50. Unlike *NER*, when the same experiment is done using *FQR*, the grant time is 100 instead. If the event with timestamp 50 incurs a rollback procedure and causes an event with timestamp less than 100 to be sent out again, the event is considered to have an invalid federation time.

Table 5.5.1: Comparison between NER and FQR in an example

NER		***FQR***	
A	B	A	B
NER(100) RAV(50) TAG(50)	NER(200)	FQR(100) RAV(50) TAG(100)	FQR(200)

Some other RTI software, such as the Federated Simulations Development Kit (FDK) [FPF00], has corrected this error, and it has also been corrected in the IEEE 1516 specification [IEEE1516.1]. For the DMSO RTI 1.3NG, however, the source code of the RTI implementation cannot be accessed to address this problem. One solution is to provide a suitable cutoff time which is used by the RTI to calculate a genuinely safe grant time in *FQR*. Since the global information cannot be known including the transient messages and the messages already in the buffer before the *FQR*, it is difficult

to provide a cutoff time other than using the last grant time. However, always using the last grant time will bring other problems for fossil collection; that is, without increasing GVT, fossil collection cannot be carried out.

In [VAM99][VAM00], the calls to the *FQR* method are intercepted by the rollback controller and executed in two phases. Initially the rollback controller uses a conservative approach, calling *NER* to receive the TSO messages from the RTI and then the federate's logical time is granted to t_f through the *TAG* callback. Since the granted time as a result of *NER* is a safe time as the federate's logical time, the time t_f can be considered as the GVT value and can be used to reclaim memory. After this conservative phase, the manager calls the *FQR* method. Using this approach, fossil collection can be processed safely. However, it will reduce the benefits of the optimistic approach since the conservative phase is interleaved with the optimistic phase every cycle.

An algorithm is proposed to solve the problem while trying to keep the characteristics of the optimistic approach. In the RTIAmbassador+, the *FQR* is extended in the middleware. The cutoff time the federate provides to the *FQR* is replaced by the last grant time in the *FQR* service. This value is always a safe guarantee for future events from this federate. But in addition, *NERA* is used to advance GVT for fossil collection periodically. Based on this property, this time advance algorithm is named FPN (*FQR* and periodical *NERA*) and the algorithm in [VAM99][VAM00] is referred to as *NF* (1 *NERA* followed by 1 *FQR*). Obviously one important issue is: when the *NERA* should be issued and how frequently this should be done.

Figure 5.5.1 shows the extended FQR in the RTIAmbassador+. For the users to construct optimistic federates, they still use the next local event time to issue a *FQR*. However, the rollback controller will process additional operations on behalf of the federate. An upper limit is set for the length of the *FederateState* queue based on the available memory. In each *FQR* procedure if this upper limit is reached, an NERA request with the next local event time will be issued by the rollback controller. Before that, a *startNERA* trigger (via a *RO* interaction) is broadcast to all other optimistic federates in the federation. Each federate receiving such a signal will invoke an *NERA* request in its next request to advance time. The federate with the smallest next local event time will be granted its request by the RTI at once. To quickly free other federates from *NERA* block waiting for a *TAG* callback, an *endNERA* trigger (via a TSO interaction) is sent out to those blocking federates, which causes the RTI to send a *TAG* to those federates. After receiving the *TAG*, the rollback controller can use the new grant time to perform fossil collection. Using this algorithm, the time spent in the *NERA* request will be quite short and meanwhile the federation time will advance safely.

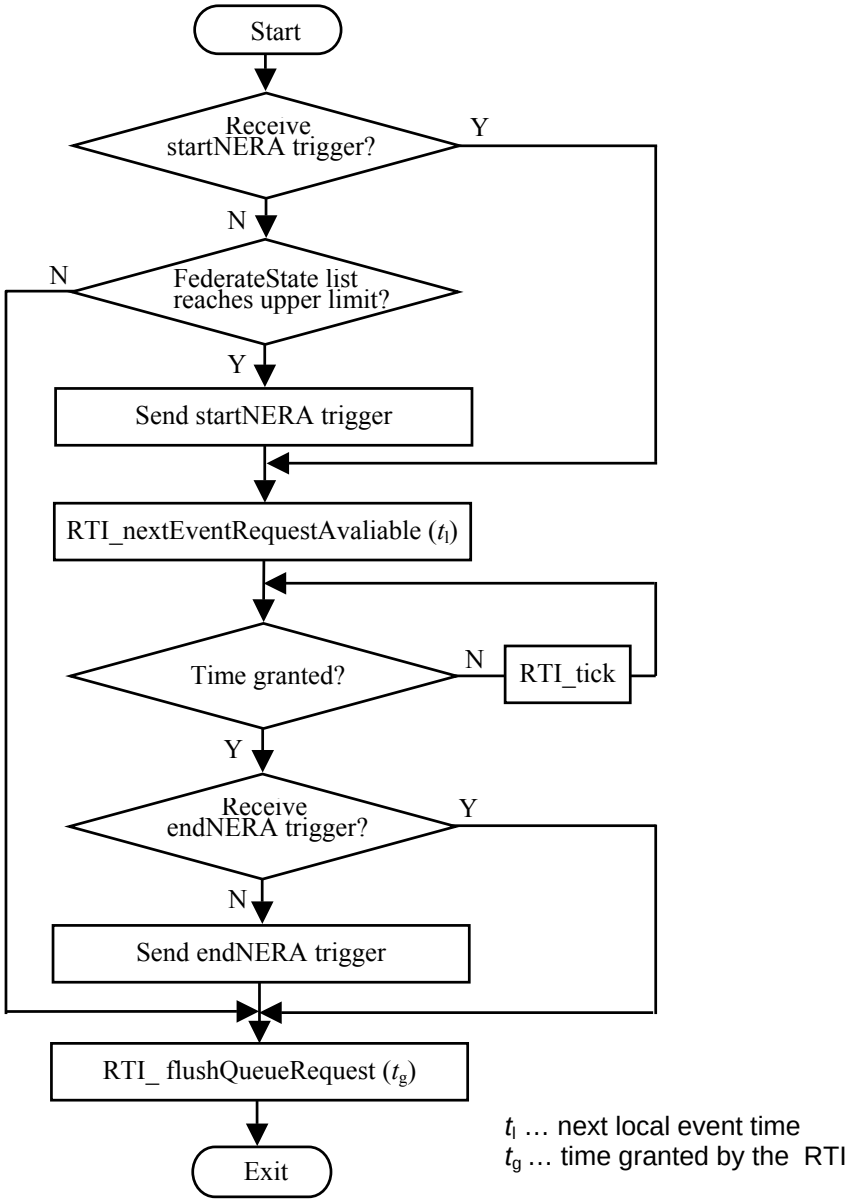


Figure 5.5.1: Extended FQR

5.6 Experiment Design and Results

The comparison between conservative and optimistic synchronization approaches is made based on the implementation of the rollback controller. A typical Type II IRM is chosen since it may demonstrate the benefits of optimistic synchronization compared to conservative synchronization when lookahead is zero or near zero.

5.6.1 Experiment Design

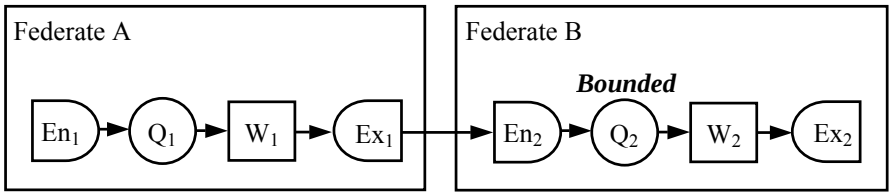


Figure 5.6.1: Bounded buffer distributed simulation model

The experiments are designed to test the rollback controller and time advance algorithm, as well as to make a comparison between the conservative and optimistic synchronization approaches using the HLA. A typical Type II IRM is used which represents some practical real-world cases, not only in manufacturing simulation but also in other application areas such as the aviation model described in [WIE98] [WIE99]. As shown in Figure 5.6.1, the federation is composed of two federates. Federate A generates entities in workstation W_1, and sends them periodically via exit point Ex_1 and entry point En_2 into a bounded queue Q_2 in federate B. When the queue becomes full, a NZL message (near zero lookahead message as discussed in Section 3.4.2) is sent back to federate A, which causes federate A to stop transferring entities to federate B. At some later time, when the entity is processed in workstation W_2, federate B clears a slot in Q_2 and sends a second NZL message to federate A, which allows new entities to be transferred. Such NZL events are a constraint to conservatively synchronized simulations since the lookahead of federate B is set to near zero. It is expected the optimistic approach to outperform the conservative approach in the situations where the queue is seldom full.

The platform for the experiments is three DELL 2.2 GHz P4 1024 MB memory computers connected via 100Mbps Ethernet running Windows XP: one is used to run rtiexec (DMSO RTI1.3NG-V6) and the other two for the two separate federates. The termination time for each federate is 100,000 simulation time units. In federate A, Q1 is not bounded and each enity is processed in W1 for a random simulation time between 100~190 time units. In each loop, after finishing processing in federate A, the entity is sent to federate B if the queue Q_2 is not full. In federate B the entity is processed in W_2 where the simulation time for processing is constant. The whole simulation is terminated when both of the two federates reach the termination time. In each federate, the spin loop for processing an event is set to 0.14 second approximately to represent the computation time.

5.6.2 Experimental Results and Analysis

Figure 5.6.2 shows the execution time versus the processing time in federate B for different maximum queue lengths of Q_2 in federate B. For the optimistic approach, two algorithms, *NF* and the new *FPN*, are used when issuing the request to advance time. Here the upper limit for the FederateState queue is set to 200, which means the *NERA* request will be issued when the length of the FederateState queue reaches 200.

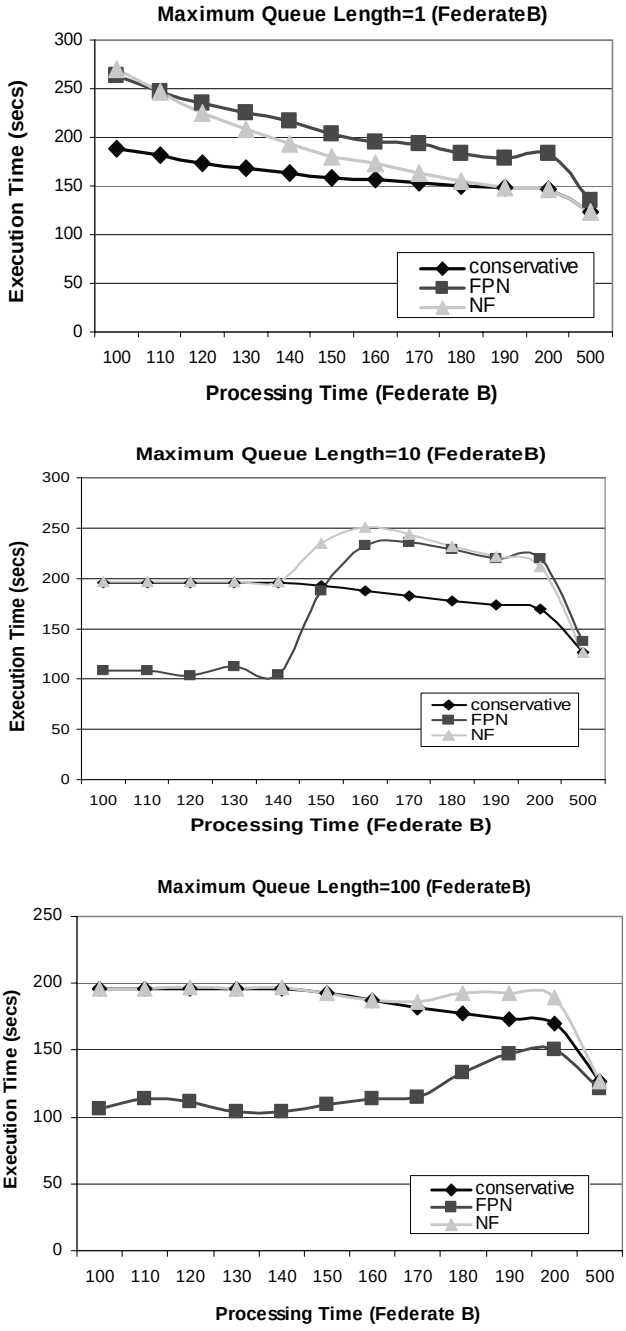


Figure 5.6.2: Execution time for conservative and optimistic synchronization approaches

As it can be seen, when the queue length is small, the conservative approach outperforms the optimistic approach. The reason is that the queue is almost always full and many NZL messages are issued, which incur many rollbacks in both federate A and federate B. However, with an increasing queue length, the performance of the new optimistic synchronization algorithm *FPN* is better than the conservative approach. This improvement can be attributed to the fact that parallelism is fully exploited by the optimistic approach. In the conservative approach, near zero lookahead means the performance of the distributed simulation is similar to that of a sequential simulation. However, in the optimistic approach, the two federates can process the computations in parallel and achieve good performance in situations where the queue is seldom full. As shown in Figure 5.6.3, in some cases the number of rollbacks is very small, even zero, which will benefit the optimistic approach.

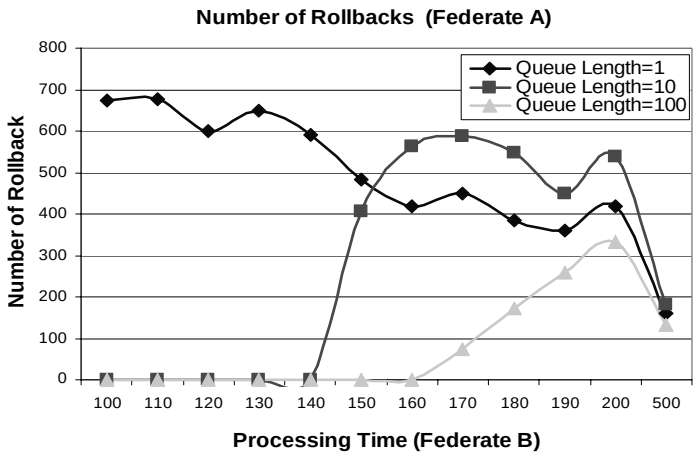


Figure 5.6.3: Number of rollbacks in federate A for different queue lengths in federate B using FPN

It is also observed that the performance of the optimistic approach with the *NF* algorithm is even worse than that of the conservative approach in many situations. It is due to the fact that this algorithm introduces some conservative elements to block the execution process while still having the optimistic overhead. The cost of the optimistic approach is mainly the cost of state saving, rollback and the anti-message overhead. That is why it is essential to utilize efficient algorithms to handle state saving, rollback and fossil collection.

By varying the upper limit of the length of the *FederateState* queue, another set of experiments is conducted for the optimistic synchronization approach with the FPN algorithm. As shown in Figure 5.6.4, the performance is also related to this value. On one hand, the longer the *NERA* interval, the more optimistic the simulation. On the other hand, if the *NERA* interval is too long the fossil collection cannot be carried out, which will introduce more memory overhead. Short *NERA* intervals can also throttle the optimism to prevent the federate from executing too far ahead into the future, which can lead to inefficient execution. Therefore, this value should be selected based on the available memory resource and some specific properties of the simulation models.

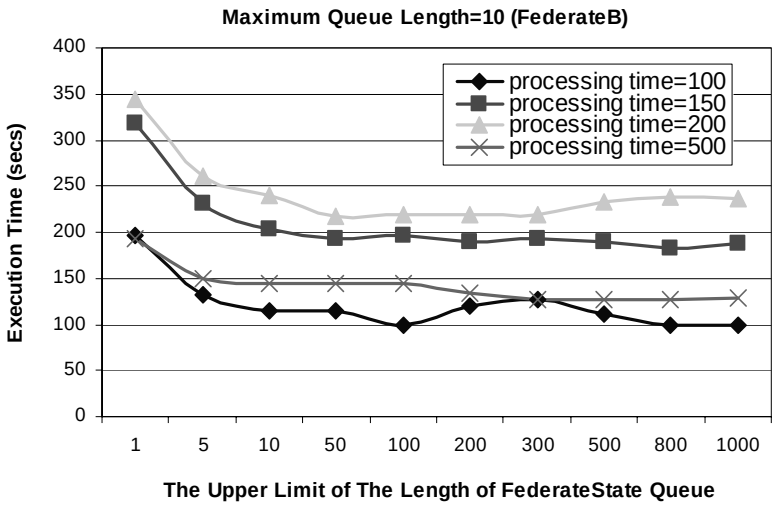


Figure 5.6.4: Execution time for optimistic approach with FPN algorithm

5.7 Scalability Study

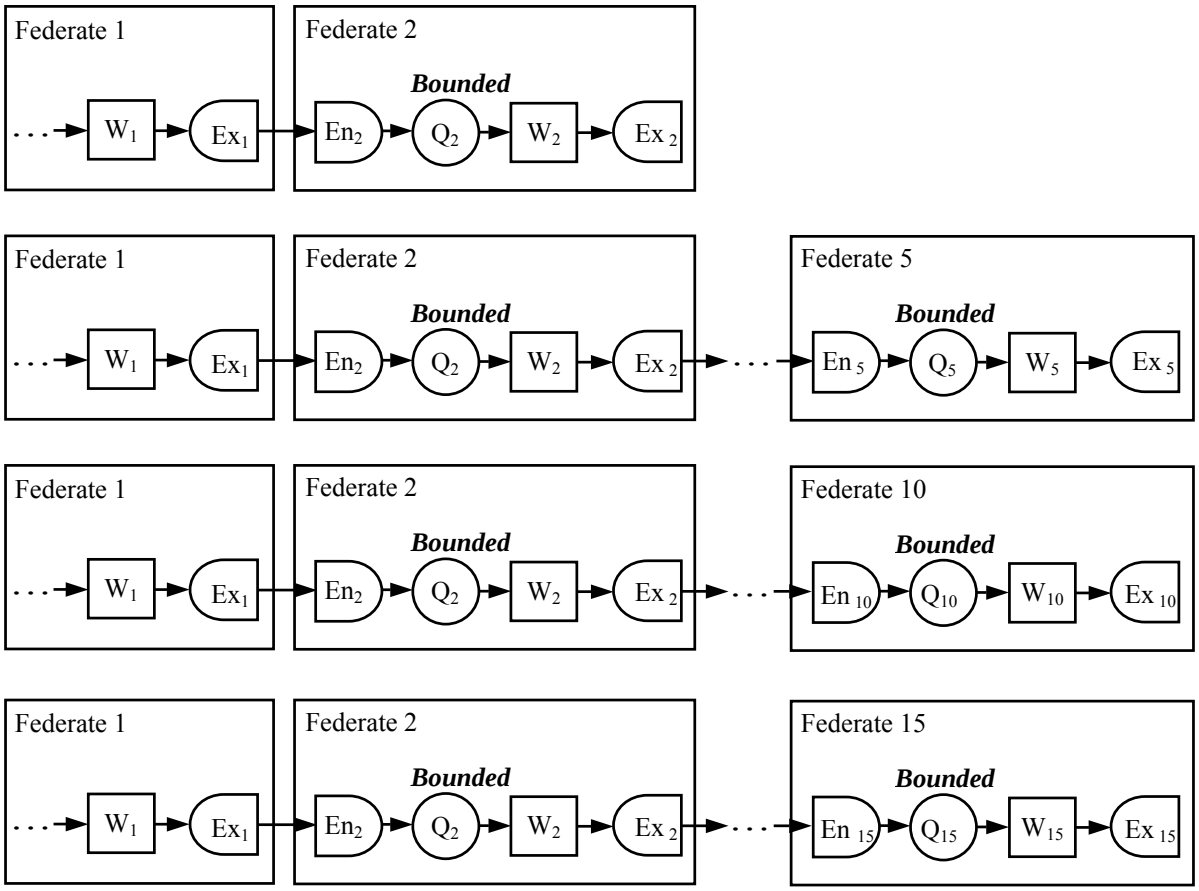


Figure 5.7.1: Bounded buffer distributed simulation model with different numbers of federates

Based on the bounded buffer distributed simulation model, five sets of experiments are carried to study scalability. As shown in Figure 5.7.1, the federation is composed of 2, 5, 10, and 15 federates respectively, organized as a production line. Federate 1 generates an entity each loop if queue Q_2 is not full and increments the simulation time by a random value between 100~190 representing the processing time of W_1. Each other federate has a bounded queue with a maximum length of 10 and a workstation with a constant processing time. Each federate processes the entity received from its predecessor and then passes it to its successor. The final product is the entity that is produced by the last federate. The whole simulation is terminated when all the federates in the federation reach the termination time. As discussed in section 5.6.1, since there are many NZL messages exchanged among the federates, the lookahead has to be set to near zero in the experiments. The platform for the scalability experiments is 16 DELL 2.4 GHz P4 512 MB memory computers connected via 100Mbps Ethernet running Windows 2000: one is used to run ritexec (DMSO RTI1.3NG-V6) and the others for the individual federates separately.

Table 5.7.1: Five sets of experiments with different processing time in the federates

	Number of federates	Description	Number of entities produced
Set 1	2	Federate 2 processes each entity for 100 simulation units.	3463
	5	Federate 2~5 processes each entity for 100 simulation units.	3461
	10	Federate 2~10 processes each entity for 100 simulation units.	3457
	15	Federate 2~15 processes each entity for 100 simulation units.	3454
Set 2	2	Federate 2 processes each entity for 170 simulation units.	2940
	5	Federate 2~5 processes each entity for 170 simulation units.	2937
	10	Federate 2~10 processes each entity for 170 simulation units.	2932
	15	Federate 2~15 processes each entity for 170 simulation units.	2926
Set 3	2	Federate 2 processes each entity for 1000 simulation units.	499
	5	Federate 2~5 processes each entity for 1000 simulation units.	496
	10	Federate 2~10 processes each entity for 1000 simulation units.	491
	15	Federate 2~15 processes each entity for 1000 simulation units.	486
Set 4	5	Federate 2~5 processes each entity for 100, 150, 200, 250 simulation units respectively.	1997
	10	Federate 2~10 processes each entity for 100, 150, 200, 250, 300, 350, 400, 450, 500 simulation units respectively.	995
	15	Federate 2~15 processes each entity for 100, 150, 200, 250, 300, 350, 400, 450, 500, 550, 600, 650, 700, 750 simulation units respectively.	659
Set 5	5	Federate 2~5 processes each entity for 250, 200, 150, 100 simulation units respectively.	1997
	10	Federate 2~10 processes each entity for 500, 450, 400, 350, 300, 250, 200, 150, 100 simulation units respectively.	995
	15	Federate 2~15 processes each entity for 750, 700, 650, 600, 550, 500, 450, 400, 350, 300, 250, 200, 150, 100 simulation units respectively.	659

By varying the processing time in each federate, five sets of experiments (listed in Table 5.7.1 above) are obtained. These are designed to compare the execution time for conservative and optimistic synchronization approaches and the number of rollbacks for the optimistic approach is also measured for analysis. These sets represent different cases which may benefit the conservative or optimistic approach. In set 1, 2 and 3 all the federates except federate 1 have the same processing time. In set 1, the queues are almost always empty since each entity can be processed quickly with a short processing time, which will favor the optimistic approach. Set 2 uses a processing time which is the worst case for the optimistic approach (as shown in the previous experiments in Figure 6.6-4) because the queue in federate 2 is usually full and a large number of entities are finally produced. Similar to set 2, the queue in federate 2 is usually full in set 3 but fewer entities are produced since the processing time is longer. In set 4 the processing time increases from federate 2 to the last federate and by contrast in set 5, it decreases from federate 2 to the last federate.

The execution results for the five sets of experiments are shown in Figure 5.7.2. With the increase of the number of federates, the optimistic approach shows better performance than the conservative approach. In the conservative approach, the performance of the distributed simulation with near zero lookahead is no better than that of a sequential simulation. It is especially worse when there are a large number of federates since nearly all the federates have to run one by one. In the optimistic approach, however, parallelism can be fully exploited. Even though the computation time in each federate will increase because some computations have to be redone, the whole simulation can run concurrently and gain a faster speed than that of the conservative approach.

The number of rollbacks in the last federate is also collected for analysis in Figure 5.7.3. There is almost no rollback in set 1 since the queues are seldom full which will always benefit the optimistic approach. In set 2 and set 3, the processing time in the federates (from federate 2 to the last federate) is longer and it will cause the queue in federate 2 to be usually full. That means some optimistic computations should be cancelled and reprocessed. The number of rollbacks is largest in set 2 because of the large throughput. It should be noted that many rollbacks are also generated in other federates due to the retraction messages sent by federate 2. In set 4, the processing time increases from federate 2 to the last federate, which leads to a full status periodically for the queues in those federates. As the reverse of set 4, the processing time decreases from federate 2 to the last federate in set 5 which makes the queue full periodically only in federate 2 and the number of rollbacks in the last federate is quite small.

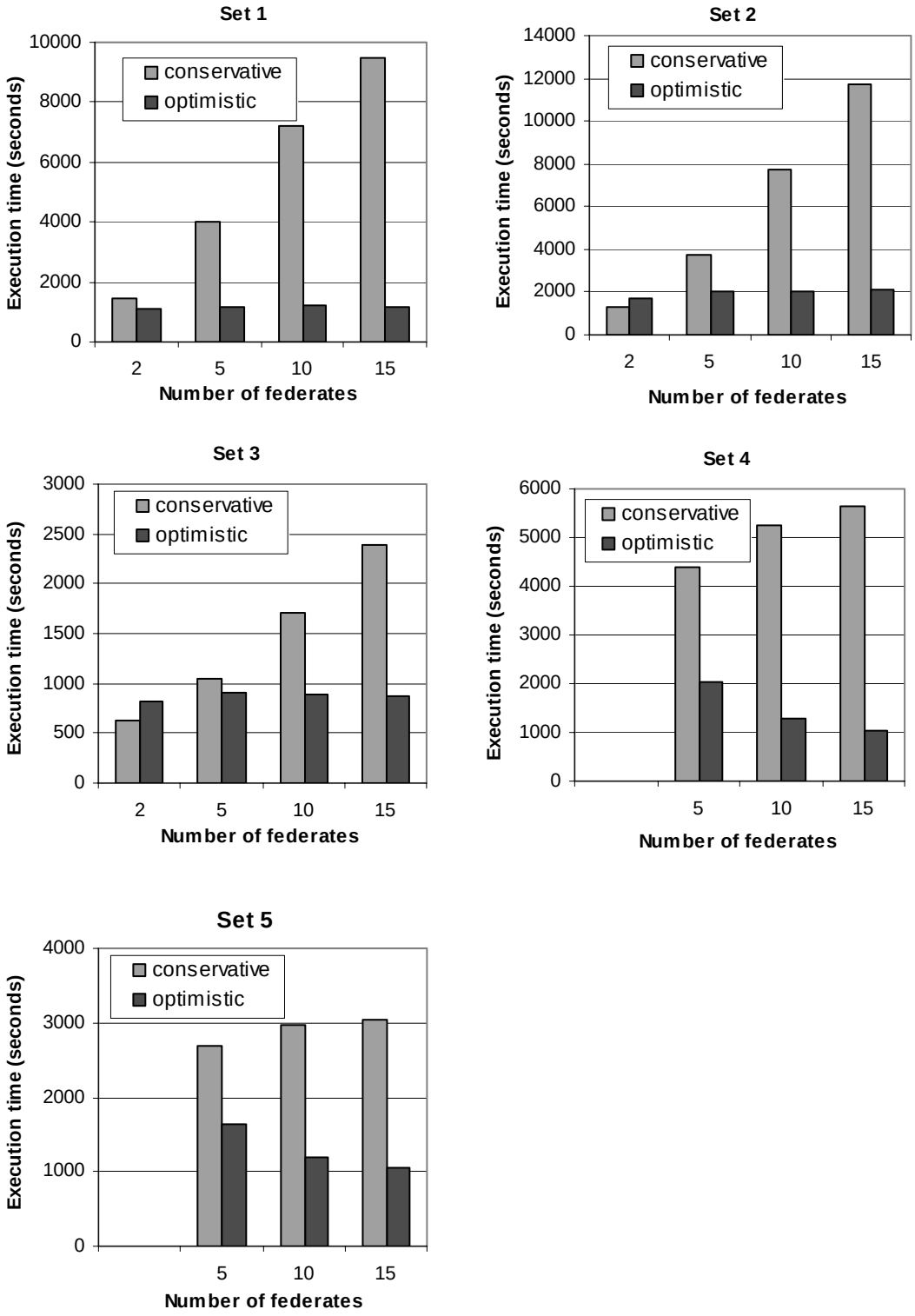


Figure 5.7.2: Execution time of five sets for both conservative and optimistic synchronization approaches

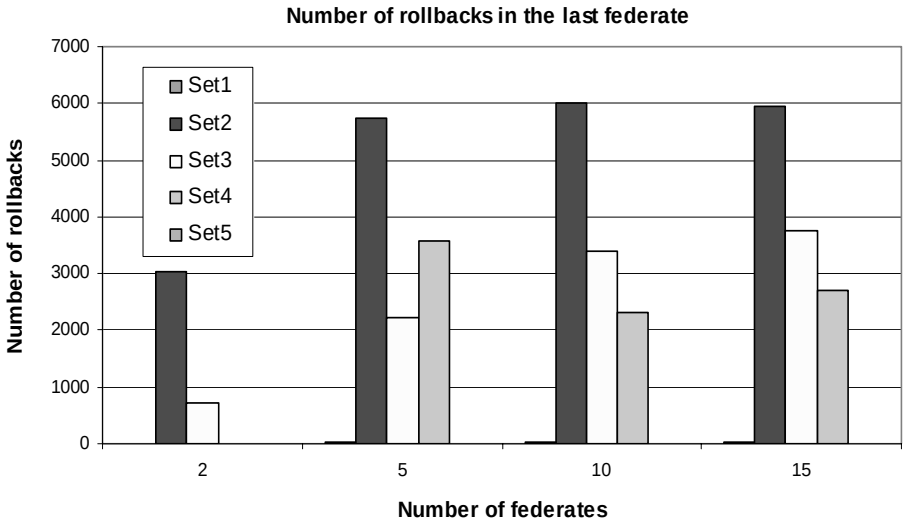


Figure 5.7.3: Number of rollbacks of five sets for optimistic synchronization approach

In summary, the results of the scalability study show that the optimistic approach can still achieve better performance for the bounded buffer distributed simulation model even in those cases where the queue is full periodically. Compared to the optimistic approach, the conservative approach is greatly constrained by the near zero lookahead and the performance is no better than that of a sequential simulation.

5.8 Summary

This chapter investigates optimistic synchronization in HLA based distributed simulations. The experimental results show that the optimistic approach can achieve better performance than the conservative approach when lookahead is difficult to exploit or in situations where causality errors may occur but do not occur. In addition, although the optimistic approach introduces more overheads from the state saving and rollback procedure, good performance is still obtained. Especially when the computation time (real time) in the model is large, the extra time for the rollback procedure only takes up a very small portion of the total time. The scalability study also shows the optimistic approach can achieve better performance due to the full exploitation of parallelism. Moreover, the proposed middleware can relieve the model developers from the burden introduced by the optimistic approach, so that the rollback controller can be embedded in the integration of the CSP with the HLA.

CHAPTER

6

SHARED STATE IN CONSERVATIVE SYNCHRONIZATION

6.1 Motivation

As discussed in the previous chapters, lookahead is the key factor for a simulation using the conservative synchronization approach to gain good performance. In the HLA, each federate regulates the time of the federation with a lookahead value. This lookahead determines the next earliest simulation time when the federate will potentially send out an interaction or object update. Hence, federates that are constrained cannot progress beyond the minimum of the current time of regulating federates plus the corresponding federate lookahead, which is referred to as the lower bound on timestamp (LBTS). This ensures that the constrained federate will not receive any update or interaction in its past. How asynchronously federates can progress in time is thus very much dependent on the lookahead value. A relatively larger lookahead value means less constraint to the federation.

However in some simulation models lookahead has to be sent to zero or near zero (see section 1.2.3) which will reduce the execution speed of the whole simulation. This is also a typical problem in CSPI-PDG Type III and Type V IRMs. In the Type III shared resources IRM (as shown in Figure 6.1.1), resource R may be a repair man that is shared between the two models. If W_1 breaks down, it will be repaired by R according to a workstation repair distribution. If W_2 also breakdowns when R is utilized by W_1, it must wait until R is released by W_1. The state information of R should be visible to M_1 and M_2 for enquiry. In the Type V shared data structures IRM (as shown in Figure 6.1.2), the data structure D is shared between M_1 and M_2. It is a more general model than the Type III IRM and can be used to represent any data that needs to be shared between the models, such as shared variables.

The implication of any distributed simulation solution to this problem is that any update to one copy of D has to be synchronized with all copies of D. The reason why the Type III IRM is also introduced is that it represents a special case which may allow an optimized approach. A general solution for the Type V IRM will also be applicable for the Type III IRM. Both of these two types of IRMs need to share some kind of state information in the distributed simulation, which is likely to introduce zero lookahead.

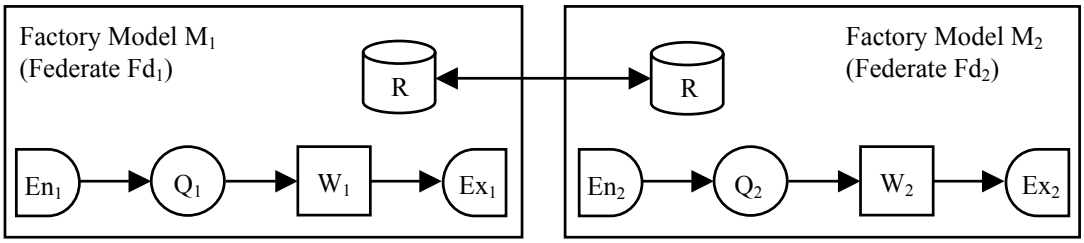


Figure 6.1.1: IRM Type III: shared resources

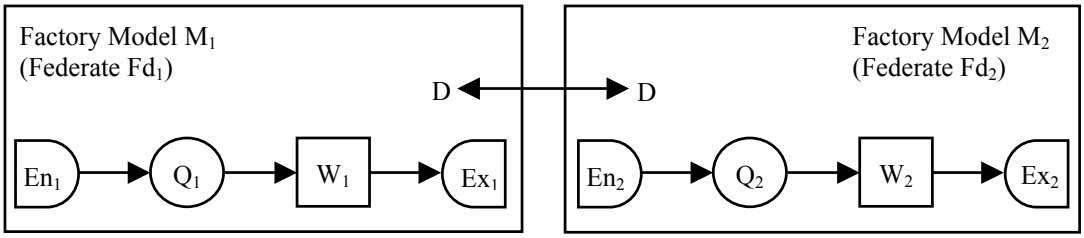


Figure 6.1.2: IRM Type V: shared data structure

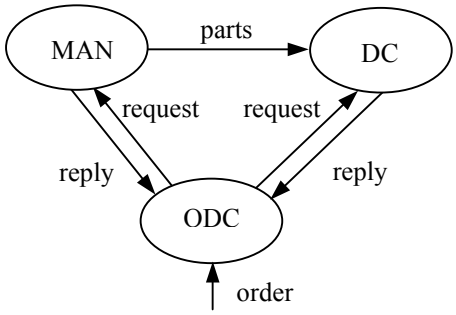


Figure 6.1.3: A supply chain simulation

This chapter investigates the problem of shared state in the simulation model, where the state variables have to be written and read at the same instant of simulation time. Suppose the state variable is updated by M_1 at time *t*. To enable the new value to be visible to M_2 instantly, an event carrying the state information should be sent by M_1 at *t*, which introduces zero lookahead in M_1. For

example, in a supply chain simulation as shown in Figure 6.1.3 there may exist some shared inventory information in the manufacturer (MAN) and the distribution center (DC) that needs to be made visible to the order dispatching center (ODC) for order dispatching purposes. Inventory information is thus held as shared state in the MAN and DC, resulting in zero lookahead.

In [MEH93] two general approaches are suggested to implement shared variables for a conservative protocol. The first approach using query events is sometimes referred to as "pull processing" because each federate is responsible for "pulling" the information it needs. The second approach called "push processing" is where the federate automatically "pushes" the value of the required state information to other federates whenever the variable changes.

This work[7] is based on these two processing methods and the HLA/RTI APIs. Two algorithms are proposed, namely *pullRO* and *pushRO*. In *pullRO* or *pushRO*, some of the timestamp order (TSO) messages that cause zero lookahead values are replaced with receive order (RO) messages. This removes the time constraint that these messages impose on the lower bound on timestamp (LBTS) calculation, which in turn will improve the time advancement rate of federates. To free the users (simulation developers) from the complex details of the implementation, a middleware approach is introduced to extend the low-level services provided by the HLA/RTI. Meanwhile, the original semantics of the RTI APIs still remain for easy use by the simulation developers.

6.2 The pullRO / pushRO Algorithms

Our approach assumes that the shared state is only updated locally and no remote write is allowed. The owner can thus go ahead of the requester in simulation time as no remote update to the state variable needs to be synchronized. This assumption is similar to the work in [LLG98b]. This is a special case of shared state but can represent many practical models, e.g., inventory information that is maintained by a local model and read by external models.

As discussed, the shared state could introduce zero lookahead. In HLA, lookahead is chosen by considering all the timestamp-order (TSO) interactions and object updates within a model (TSO interaction/object updates are delivered in timestamp order to the receiving federates). The smallest time increment required by a TSO interaction/object update is chosen as the federate lookahead. To reduce this constraint, such TSO object/interaction update messages are replaced with RO messages and the time information is held as one attribute/parameter. By doing so, a larger lookahead can be

[7] This work is carried out as part of a collaborative research project between NTU and SIMTech on "Technology for Secure and Robust Distributed Supply Chain Simulation".

obtained by not considering the TSO messages that result in zero lookahead in the lookahead computation.

Whatever algorithm is used, one rule must be obeyed, that is, to keep causality. Since RTI does not provide such a constraint on RO messages, an alternative mechanism should be adopted. This mechanism is realized by introducing a history list to the owner federate in the pull approach and a future list to the requester federate in the push approach. The two lists keep all changed values for each state variable, associating with each value the simulation time at which the state variable is updated. The requester can obtain the value of the state variable through these lists if a valid value exists. Otherwise, the requester needs to wait until a valid value becomes available. So in this mechanism, the causality of the simulation is preserved. It is relying on the requester not to move forward in simulation time when its request cannot be satisfied. When the requester does not move forward, it in turn constrains the owner from progressing since it is possible there are some other TSO messages sent from the requester to the owner. As this synchronization only happens when the requester needs a value of the state variable and those TSO messages incurring zero lookahead are replaced with RO messages, it potentially improves the simulation speed.

In the *pullRO* approach, the owner needs to keep a history list to guarantee that the requester can obtain a valid value of the state variable when needed. Once a requester issues a request to the owner at t_r, the owner will search through its history list to look for the values with update times satisfying the following condition: $t_l \le t_r < t_u$, where t_l and t_u are the two adjacent update times of the state variable. This means the owner has passed the simulation time t_r and the update at time t_l is taken as it is the latest update before the time of the request. Otherwise, the request is recorded and the reply is not issued until the owner moves beyond t_r. In this algorithm, the requester could achieve the correct value just by sending RO messages for requests.

In the *pushRO* approach, the owner is also allowed to run faster than the requester and a future list is used in the requester to store future values. Before the requester issues a request at the simulation time t_r to the owner, it first searches through its local future list to look for values with update time satisfying the condition: $t_l \le t_r < t_u$, where t_l and t_u are the two adjacent update times of the state variable. If such a value can be found, the value updated at t_l is returned as the result. Otherwise, this request is sent out to the owner. The *pushRO* approach will then work just as the *pullRO* approach stated before.

Another important mechanism behind these two algorithms is the fossil collection which is somewhat similar to that in the optimistic approach. Here the fossil collection is used to discard old values of shared state that are no longer needed. This is critical as the growing history list and future list will

take up too much memory which in turn might affect the performance of the simulation. For the *pullRO*, the owner can discard any shared state values in its history list that have an update time smaller than t_c, where t_c is the minimum federate time in the whole federation. This information can be obtained easily by requesting for the federate time of all federates from the MOM (Management Object Model) (see [IEEE1516.2] Section 3.1.50). As for the *pushRO*, it is very straightforward. Any shared state values in its future list that have update time smaller than t_c, where t_c is the current federate time of the requester, can be discarded.

6.3 The Implementation

To relieve the simulation developer from the burden of handling details about the above algorithms, a middleware approach is provided to extend the original services in the RTI. All the services the federate calls will be intercepted by a *state manager* in the middleware[8] (see Figure 6.3.1), which implements the state management on behalf of the federate. The state manager has corresponding member functions to add a new state variable, record a new state value, search a state value at a specified time and discard unneeded state values. It is important that all the processes are transparent to the simulation developers. They can still build their simulation federate as before, but instead of linking the RTI library to their simulation program, the middleware library RTI+, is linked (as shown in Section 3.1). Adopting this approach, the only thing that the user needs to do is to issue the request with a specified time. All the updates and requests are still carried out as before, and the translation of TSO related calls to RO related calls (combining time information) is transparent to the user.

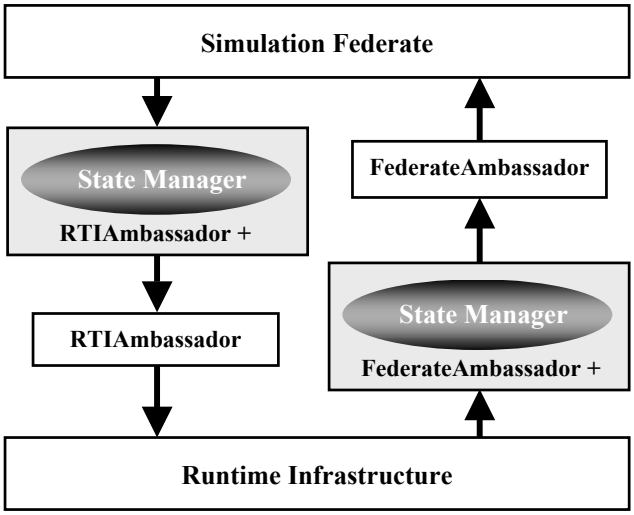


Figure 6.3.1: Middleware approach

[8] This middleware was implemented jointly by members of the project team, including Dr. Wei Junhu, Mr. Gan Boon Ping and the author. The author mainly focused on the implementation of the extended RTI.

In our algorithms, the requester needs to issue a request for an object or class update at a specified simulation time. However, in the original RTI, only *requestObjectAttributeValueUpdate* and *requestClassAttributeValueUpdate* methods without the time argument are available. This means a requester will only receive an update at a time when the owner receives the request, rather than at a time that the requester needs the value. Similar to the two methods, two new methods are created which take time as an argument. Figure 6.3.2 below shows the API of these methods. Both methods translate the call to a TSO system interaction that is sent as a request to the owner of the state variable. This interaction contains the object/class handle that the requester is requesting, and the simulation time when the value is needed.

```
void requestObjectAttributedValueUpdate ( ObjectHandle theObject,
                                          AttributeHandleSet theAttributes,
                                          RTI::FedTime theTime)

void requestClassAttributedValueUpdate ( ClassHandle theClass,
                                         AttributeHandleSet theAttributes,
                                         RTI::FedTime theTime)
```

Figure 6.3.2: New services in shared state APIs for RTI

6.3.1 The pullTSO Approach

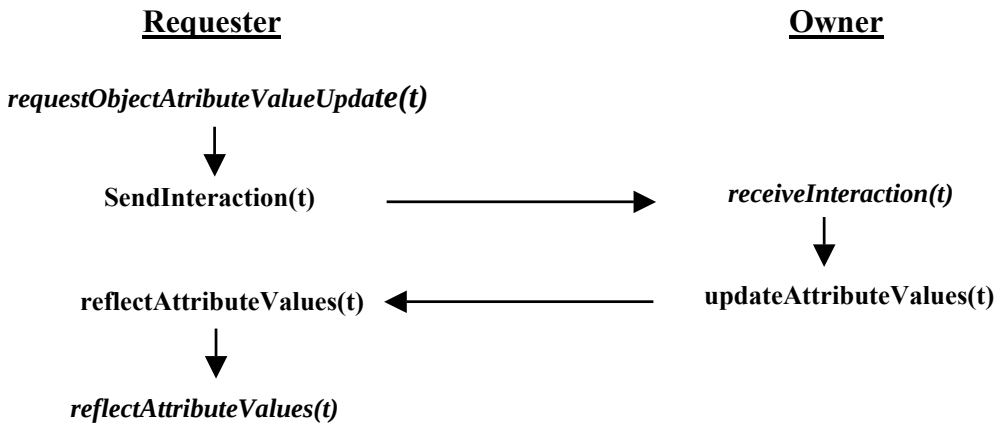


Figure 6.3.3: Sequence of calls for requesting object update – the TSO approach

Figure 6.3.3 illustrates the mechanism of the *pullTSO* approach which will be the benchmark to compare with the new algorithms. A call to the request method in the middleware is translated to a TSO *sendInteraction* call in the original RTI ambassador at the requester side. A *receiveInteraction* callback is triggered in the middleware's federate ambassador at the owner side, which in turn triggers the owner to reply through a call to *updateAttributeValues* of the original RTI ambassador. One important point is that the *receiveInteraction* is not allowed to call the *updateAttributeValues*

within the FederateAmbassador+. Hence, the middleware needs to store the request, and initiates the reply phase once the control is returned to the RTIAmbassador+. During the reply phase, the owner will reply to the requester once a valid value for the requested shared state is available. Meanwhile, the requester will wait for *reflectAttributeValueUpdate* to be called before it makes any further progress in time. The naming convention that will be used in the remaining sections of this chapter is as follows: method calls in *italic* font are calls to the RTI+ library, while method calls in standard font are calls to the original RTI library. Method calls with a *t* in the bracket are TSO method calls.

6.3.2 The pullRO Approach

```
;; system interaction classes
        (class reqeustObjectAttributeUpdate reliable timestamp
                (parameter ObjectHandle)
                (parameter AttributesHandleSets)
                (parameter Time)
        )
```

Figure 6.3.4: System interaction class in .fed file

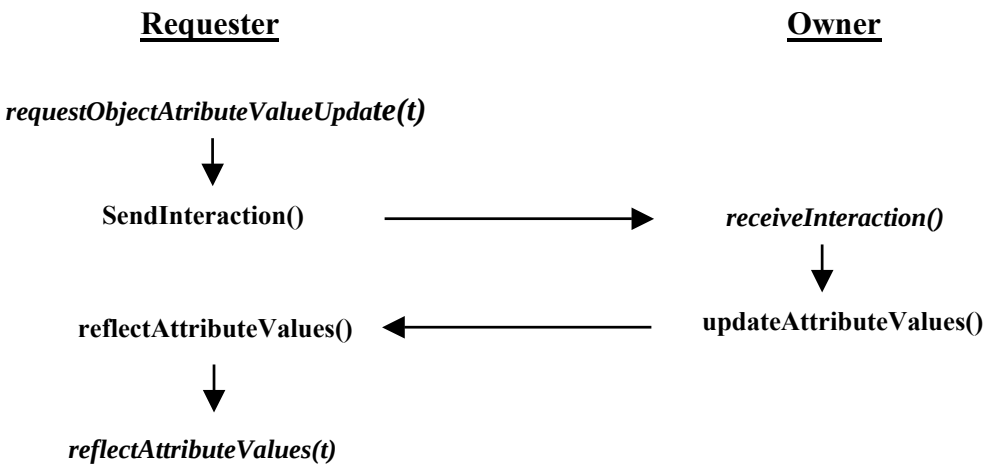


Figure 6.3.5: pullRO – sequence of calls for requesting object update

In the *pullRO* approach, the middleware will internally alter the TSO object/interaction class of shared variables to a RO object/interaction class and add the time information as an argument into the system interaction (see Figure 6.3.4). Figure 6.3.5 illustrates the sequence of method calls to implement the *pullRO* approach. The implementation of the *pullRO* approach is totally transparent to the user. The RTI+ translates the TSO request call to a RO *sendInteraction* at the requester end. On the owner side, the middleware stores each object/interaction update in its history list (and does not really send out these updates). As discussed earlier, each request is recorded, and the reply is deferred

until the control is returned to the RTIAmbassador+ of the middleware. In the reply phase, the middleware first looks through the history list to see if a valid value can be found. If a value is found, the RO *updateAttributeValues* is initiated straightaway. Otherwise, the request is processed when the simulation time of the owner moves past the request time.

At the requester end, once a request is issued, the requester is not allowed to progress in time, until it receives the update that corresponds to its request. This is realized by keeping track of pending requests, and withholding the time advance request NER until the request is received.

6.3.3 The pushRO Approach

In the *pushRO* approach (see Figure 6.3.6) all updates to the shared state are pushed towards the requester even if the requester does not need the value. On the owner side the RTI+ translates the TSO *updateAttributeValues* call to an RO *updateAttributeValues*. The middleware at the requester side will then record the object update, and associate the update time to the update. When the requester issues a request for a shared state value, the local future list is first searched. If a valid value is found, the TSO *reflectAttributeValues* is called to deliver the value to the requester. However, the user's *reflectAttributeValues* is not called immediately but is deferred until the simulation federate attempts to pass control to the RTI, as in the HLA, callback messages are only delivered to a simulation federate when the control is with the RTI. If no valid value is found, RTI+ will convert the *pushRO* approach to the *pullRO* approach (as illustrated in Figure 6.3.5) and uses the same system interaction (as illustrated in Figure 6.3.4).

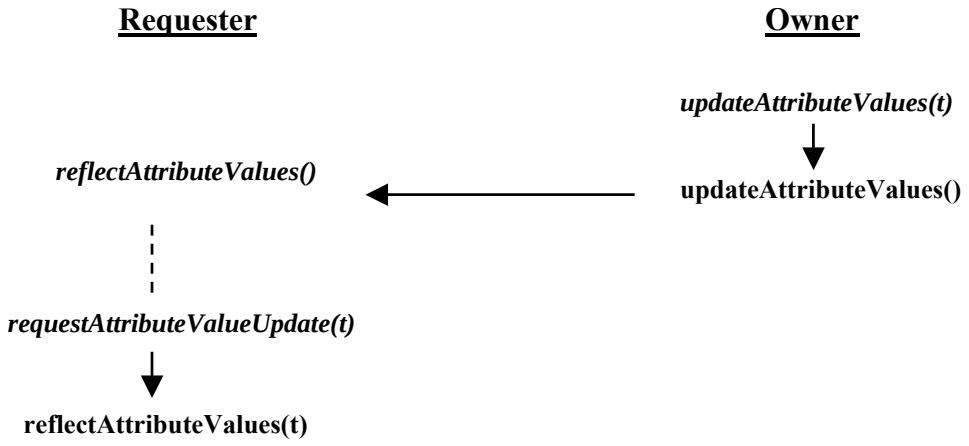


Figure 6.3.6: pushRO – sequence of calls for requesting object update

6.4 Experimental Results

The performance of the proposed approaches, *pullRO* and *pushRO*, are compared with *pushTSO* using a simple request-reply distributed simulation. The simulation consists of an owner federate and a requester federate running on two computers interconnected by 100Mbps Ethernet. The owner federate periodically updates a shared variable with a 100 time unit interval. Requests are issued with an interval of 0.1, 0.2, 0.5, 1.0, 2.0, 5.0, and 10.0 times the update interval, as this ratio will have a significant impact on the performance. Intuitively, a lower request-to-update interval ratio will favor the *pushRO* as the requests happen more frequently than the updates. On the other hand, a higher request-to-update interval ratio will favor the *pullRO* as it generates fewer requests than there are updates. Also, lookahead values of 10, 100, and 1000 are used to mimic models that have other TSO messages to be sent/updated. In the case where the request-reply is sent through a TSO message (*pullTSO* approach), the lookahead will be zero no matter what lookahead value is used, as the request-reply is carried out with zero time increment. But a non-zero lookahead can be used when the TSO request-reply messages are replaced with RO messages using the *pullRO* and *pushRO* approach. The experiments are run with 10,000 updates.

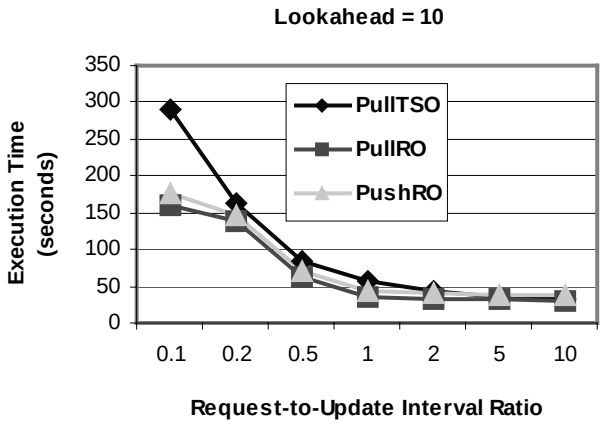


Figure 6.4.1: Execution time vs. Request-to-Update interval ratio (Lookahead=10)

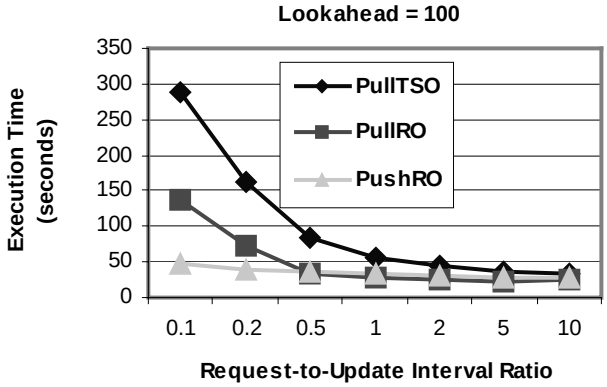


Figure 6.4.2: Execution time vs. Request-to-Update interval ratio (Lookahead=100)

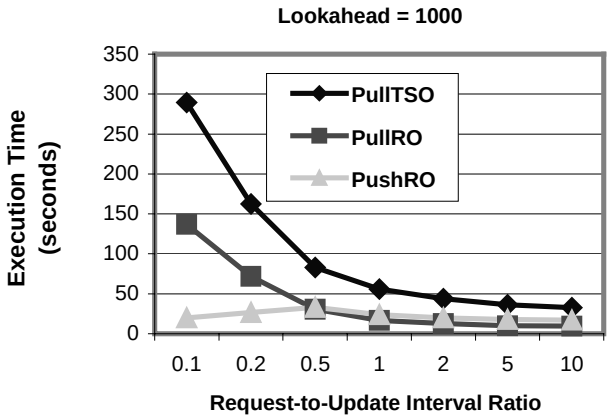


Figure 6.4.3: Execution time vs. Request-to-Update interval ratio (Lookahead=1000)

Figures 6.4.1, 6.4.2, and 6.4.3 show the execution time achieved with varying request-to-update interval ratio, for lookahead values of 10, 100, and 1000 respectively. As it can be seen, the execution time for the *pullRO* and *pushRO* implementations generally decrease as the lookahead is increased from 10 to 1000. These two approaches also perform better than the *pullTSO* approach. This improvement can be attributed to the fact that the owner can run ahead of the requester due to the larger lookahead. This is because with a larger lookahead, a larger time grant can be achieved. Whenever the requester needs a value at a specific simulation time, the value is already available either in the history list of the owner in the *pullRO* approach, or in the future list of the requester in the *pushRO* approach.

It is also observed from the figures that the *pushRO* approach outperforms the *pullRO* approach generally with small request-to-update interval ratio. Conversely, the *pullRO* outperforms the *pushRO* for large request-to-update interval ratio. This is due to the different communication overhead caused by message transmissions.

With a small request-to-update interval ratio, there is large number of requests from the requester compared with the number of updates from the owner. In the *pushRO* approach the requester firstly searches its future list and may find the state value in its future list, as the owner runs ahead in simulation time compared with the requester. While in the *pullRO* approach the requester needs to issue a number of requests that will slow down the execution speed. For example, the number of requests for lookahead of 1000 and ratio of 0.1 is only 25 in the *pushRO* approach but 100,000 in the *pullRO* approach (refer to Table 6.4.1).

Table 6.4.1: Number of the requests for small Request-to-Update interval ratio

	Request-to-Update Interval Ratio	***pullRO***	***pushRO***
Lookahead=10	0.1	100000	100000
	0.2	50000	50000
	0.5	20000	20000
Lookahead=100	0.1	100000	24441
	0.2	50000	21877
	0.5	20000	19608
Lookahead=1000	0.1	100000	25
	0.2	50000	9815
	0.5	20000	10000

With a large request-to-update interval ratio, there is small number of requests from the requester compared with the number of updates from the owner. This means that the requester is synchronized less often with the owner with a large ratio. In the *pushRO* approach, the owner needs to send out all the updates which will introduce more messages for transmission. While in the pullRO approach, the owner only needs to send out the updates when the requests are issued from the requester. For example, the number of requests for lookahead of 1000 and ratio of 10.0 is only 1000 in *pullRO* but the number of updates is 10,000 for *pushRO*.

Another trend can also be observed for a small request-to-update interval ratio. In this case, the *pushRO* approach outperforms the *pullRO* approach greatly when the lookahead is larger. Conversely, when the lookahead is small, the performance of the *pushRO* approach and the *pullRO* approach is nearly the same. This is due to the how far the owner can run ahead of the requester. As discussed, with a large lookahead the owner can go further ahead of the requester which will benefit the *pushRO* approach. However, if the lookahead is small, the time advance request from the owner can not be granted instantly and thus the new state value can not be pushed into the requester's future list. So when the requester finds the needed state value is not available, it will still issue the request. As seen from Table 6.4.1, the number of requests in the *pushRO* approach is the same as that in the *pullRO* approach for lookahead of 10. While with large lookahead, the owner can be allowed to run further ahead than the requester and update the new state values. Thus in the *pushRO* approach almost each time the requester can find the needed state value in its future list and fewer additional requests are issued. As shown in Table 6.4.1, the number of requests in *pushRO* is much smaller than that in *pullRO* for lookahead value of 100 and 1000.

6.5 Summary

This chapter proposes *pullRO* and *pushRO* algorithms to solve the zero lookahead problem introduced by shared state in the simulation model. It is implemented by replacing TSO messages that cause a zero lookahead value with RO messages. This removes the time constraint that these messages impose on the calculation of the simulation time, which in turn will improve the time advancement rate of federates. To preserve the causality of the simulation, a history list is introduced to the owner in the pull approach and a future list is introduced to the requester in the push approach. For each state variable in the lists, a time is associating with each value to indicate when the state variable is updated. In this way, the request can receive correct values and the simulation speed may be improved by exploiting the larger lookahead. To free the users from the complex details of the implementation, a middleware approach is introduced to extend the low-level services provided by the HLA/RTI. Meanwhile, the original semantics of the RTI APIs still remain for easy use by the simulation developers.

The experimental results show the *pullRO* and *pushRO* approaches consistently outperform simulations where there are zero lookahead updates/interactions. Even though the two approaches introduce the overhead of managing the history and future lists through the middleware, the experimental results show that this overhead is minimal. It is also observed that the *pushRO* approach and the *pullRO* approach can achieve different performances with different request-to-update interval ratios. In [LGW06], both approaches are enhanced to include time guarantee information to each update, such that the requester can have information on the validity duration of an update. This can help to cut down the number of requests being sent by the *pullRO* approach, and also benefits the *pushRO* as the *pushRO* approach reverts back to *pullRO* when the requester always runs faster than the owner.

Above all, the proposed algorithms provide the possibility to solve the problems associated with some kinds of simulation models where shared state exists (e.g., CSPI-PDG Type III and Type V IRMs). This work will be further extended to provide a complete solution to these IRMs that can be introduced into the generic CSP-HLA interface.

CHAPTER

7

CASE STUDIES

7.1 Introduction

In previous chapters, a description is given of the generic architecture for integrating a CSP with the HLA as well as a CSPE to investigate potential interoperability solutions. Some simple experiments are designed and performed to show the correctness of the CSPE and the validity of the DSManager middleware. It is interesting to utilize the CSPE to build more complicated and practical simulation models (e.g., manufacturing models or business process models), and to apply distributed simulation to solve some existing problems in these areas. Moreover, it also makes sense to see how the proposed generic architecture can enhance a real CSP to support distributed simulation and how complex the modifications made to the CSP must be to achieve this purpose.

In this chapter, some case studies will be presented to test the generic architecture proposed in this thesis as well as the CSPE. The first two case studies are executed based on the CSPE[9]. One is a semiconductor wafer manufacturing model. With individual wafer fabrication plants being modeled as separate simulation models, a borderless fab (fabrication factory) model [LGL04] [GLT05] is built by integrating two factory models each running on the CSPE. Our work provides good experience for the future integration of CSPE and AutoSched AP [ASAP]. The other case model is to integrate a simulation model built using the CSPE with a business application in the aircraft industry area. The CSPE model simulating flight component failures in integrated with a resource management module

[9] The two CSPE case studies are implemented as two undergraduate final year projects in Nanyang Technological University (see [ZHA05] and [KUS05]). The author extended the CSPE for their applications and helped to supervise their model design and experiments.

(business application) which includes the simulation of logistic flights and a decision making algorithm.

Different from the two models built using the CSPE, another case study[10] is carried out to enable a real CSP, IBM's WBI Modeler [WBIM], to support distributed simulation. With proprietary access to the source code and APIs of the WBI Modeler, experimental modifications are made to integrate with the DSManager. To verify the HLA-enabled WBI Modeler, some experiments are carried out using a demand conditioning process, in which mismatches between supply and demand are identified and corrective actions are initiated.

7.2 A Semiconductor Wafer Manufacturing Model

7.2.1 Motivation

Semiconductor manufacturing is a complicated process that is sensitive to many factors such as load balancing between wafer fabrication plants, machine breakdown, and the management of work in progress [GTT05]. Simulation using various CSPs (e.g., Arena, AutoMod, AutoSched AP and Witness) has been successfully applied to semiconductor manufacturing. The semiconductor supply chain often involves multiple companies across enterprise boundaries, each of which may have existing simulation models to perform "what-if" kind of analysis of its own operation [TCG01]. It is ideal to reuse these available simulation components to form a large-scale distributed simulation. Moreover, these simulation models may be created using different programming languages or CSPs and run on different hardware platforms. Other reasons, such as information hiding, performance improvement and geographical distribution, also motivate the utilization of distributed simulation in this area. The interoperation of simulation models of individual wafer fabrication plants to simulate semiconductor supply chains is proposed in [GLJ00] [LJG03]. However, it is difficult to achieve without a standardized approach for using the HLA to support the interoperability of CSPs. Based on the proposed generic architecture and the developed CSPE, the semiconductor wafer manufacturing process is built by linking two factory models as two CSPE federates.

In this case study, a borderless fab that involves two wafer fabs located in close proximity is simulated by integrating the individual CSP models. It aims to investigate interoperability of CSPs by developing a discrete event simulation model, which reflects a semiconductor manufacturing

[10] This work is carried out during the author's internship (under the supervision of Dr. Steve Buckley) in IBM T.J. Watson Research Center in USA from Jul. 5th, 2005 to Oct. 7th, 2005.

scenario. It also contributes to the work of the CSPI-PDG, by validating the existing proposals and investigating possible problems.

7.2.2 Model Analysis

The semiconductor wafer fabrication model built using the CSPE in this case study can be described as one wafer fabrication factory (wafer fab) having capability of producing ten wafer product types. The process flows of the wafer products considered range from 400 to 500 steps. A total of 119 workstation types were modeled in the fab, including the downtime behavior of each workstation. Some examples of workstations in the wafer fab are: wet benches, furnaces, steppers, implanters and metrology tools.

The wafer fabrication process is described by some files with standard data format specification, developed by Sematech [FFR94]. It consists of a sequence of steps through which wafer lots go, transforming the lot from raw material to finished products. A fixed number of wafers are released as one lot. Lots released at a given rate are categorized into different process flows according to their Process Flow ID. Lots with the same Process Flow ID are processed in sequence based on the Step ID. At each step a certain tool or workstation from one tool set specified by the Tool Set ID is claimed. Successfully claiming the workstation will result in the production lot being scheduled onto the workstation. The workstation is marked as busy, and the production lot is then delayed for a pre-defined processing time, which can be a constant value or a value generated from a random distribution. Upon completion, the claimed workstation is marked as idle and the lot leaves the current step, and moves on to the next step, where it repeats the whole process of claiming the required workstation. Failure to claim any of the resources in the process flow will cause the lot to be queued, waiting for the next workstation to be idle.

While a lot is being processed, a workstation can break down due to some unforeseen circumstances. This is modeled as random events in CSPs, defined using two parameters: mean time between failure (MTBF) and mean time to repair (MTTR). MTBF dictates the breakdown frequency of the resources, while MTTR defines the amount of time to repair the workstations. The actual value is generated from some probability distributions. Lots are preempted from the time that the workstation breaks down, until the workstation is repaired. The release time of lots from the workstation is thus delayed to a later time. It is also possible that a workstation breaks down during processing some lots.

Batch processing is also a typical characteristic in the given model. A number of lots in a queue are batched together, and the same sequence of actions is executed after a minimum batch size is satisfied or a time out has expired. The time out is defined as the maximum time that a workstation

will wait to accumulate lots for batching. Note that batching leads to the accumulation of lots processed on a workstation, the processing time therefore will increase.

7.2.3 Model Design and Experiments

7.2.3.1 Standalone Model

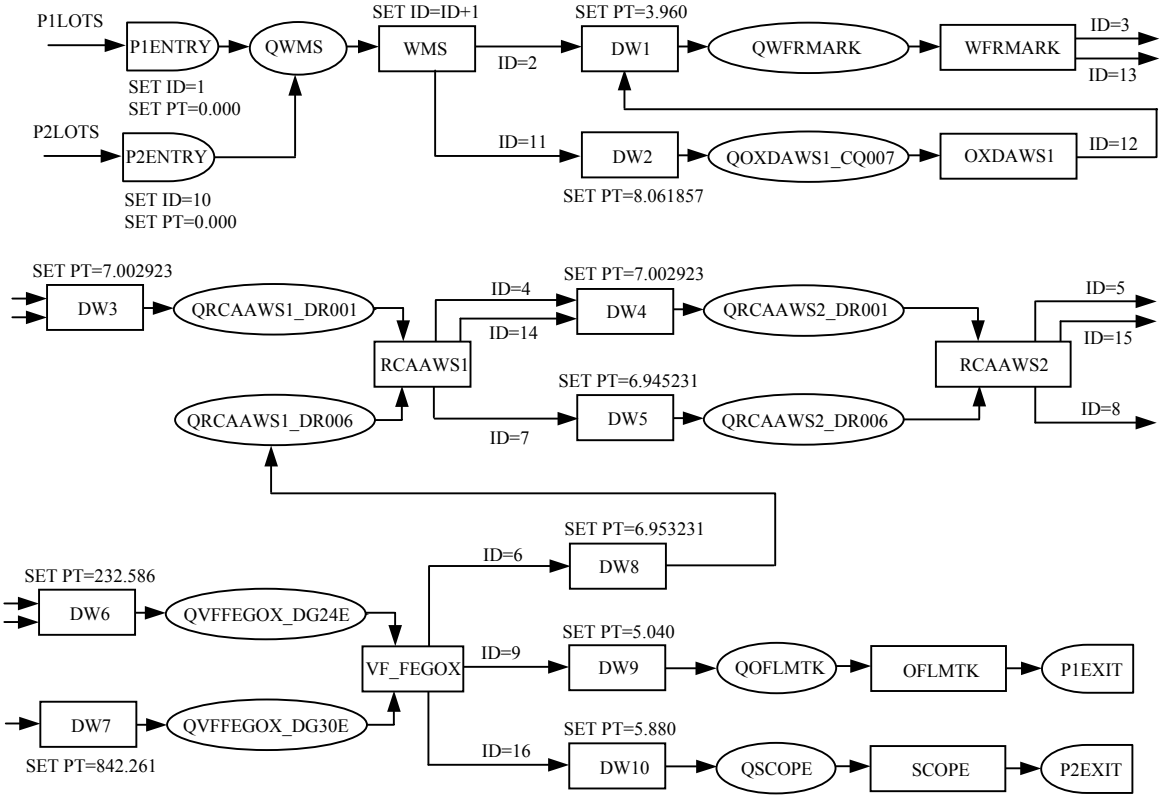


Figure 7.2.1: Small scale standalone CSPE model design

Figure 7.2.1 shows a subset of the case model represented in CSPE simulation objects. As discussed in Chapter 5, the main objective of the CSPE is to investigate and to benchmark alternative interoperability solutions. The CSPE focuses on interoperability and has limited functionality to support complicated simulation models as compared to current CSPs. However, the model in this case study imposes some requirements for advanced features in the manufacturing area. Based on the model analysis, new functionality (such as the tool set, batching rule, etc.) is provided to enable the creation of complicated manufacturing models. Another problem is to define the processing time used to process an entity or a set of entities. In the case study, the processing time is set based on the step ID of the process flow. Each workstation may be associated with different processing times for different process flows and flow steps. However, in the CSPE, the processing time associated with a workstation is defined by some distribution, or from a read-in file, or by an entity attribute value. To

solve the problem, a dummy workstation (DW) is introduced here to set the value of the attribute "PT" (processing time) based on another attribute "ID" (as shown in Figure 7.2.1). In the CSPE, an entity attribute value can be changed via an "entity action" in a workstation or an entry point and it will take effect after the entity leaves the current workstation or entry point. However, if at the next step the entities will be transferred to multiple destinations requiring different attribute values, some dummy workstations have to been added for attribute value setting before each entity arrives at its destination (as in the example shown in Figure 7.2.2).

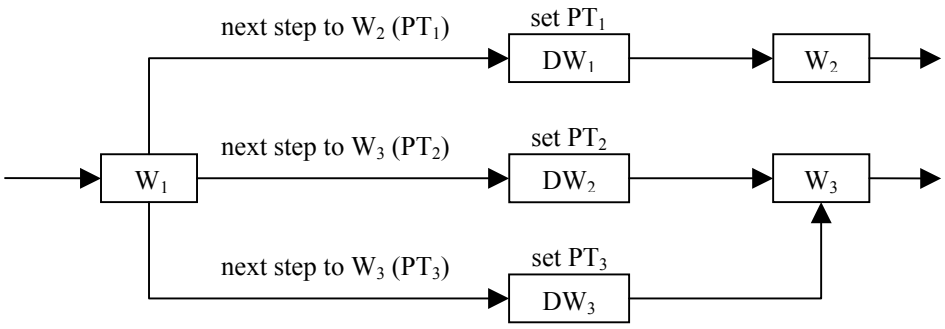


Figure 7.2.2: Processing time setting using dummy workstation in CSPE

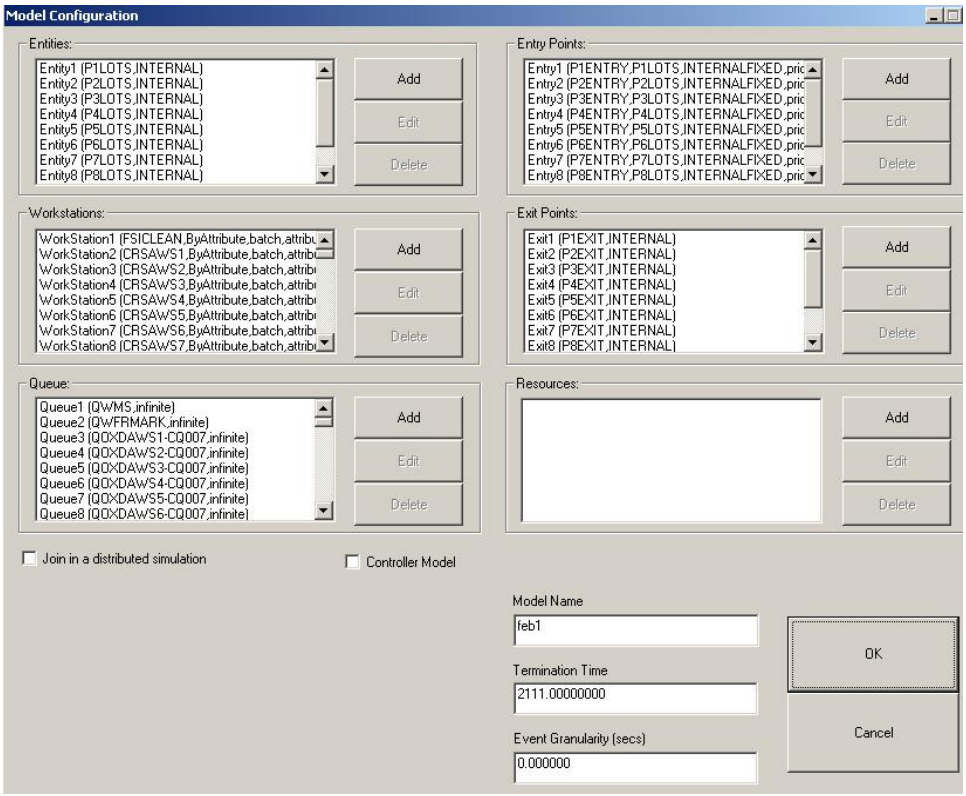


Figure 7.2.3: Standalone model shown in CSPE GUI

The CSPE supports two ways of model creation. As a straightforward way, a GUI is provided to allow the modeler to define the entity, simulation objects (entry point, workstation, queue and exit

point) and some general information (simulation time, interoperability information for distributed simulation, etc.) via corresponding GUI menus. For a complicated model with a large number of workstations or queues having similar functionality, however, it is more convenient to adopt the other approach – constructing a *.cspe file describing the structure of the model. In the case study, a conversion program is developed to transform the data files following the Sematech data format specifications into a CSPE readable file (.cspe). The generated *.cspe file is loaded by the CSPE and can then be displayed in the CSPE GUI (as shown in Figure 7.2.3).

The result of running the standalone CSPE model is validated against the same fab model created using AutoSched AP (ASAP). ASAP is a CSP supplied by Brooks Automation that is specially customized to model wafer fabrication plants. It is widely used in the semiconductor industry to answer questions such as the impact of a dispatching rule or product mix changes to the factory performance, or identifying bottleneck equipment with varying demand profiles, etc. The CSPE model is run for a simulation period of 43200, which is equivalent to 30 days in manufacturing. The number of lots collected at the exits points and the cycle times for each of the ten process flows are listed in Table 7.2.1.

Table 7.2.1: Simulation results of the CSPE model

Execution time: 6735.3039 seconds (1.8709 hours)					
Entity Type	**Lots collected**	**Cycle Time**	**Entity Type**	**Lots Collected**	**Cycle Time**
P1EXIT	14	28409.964	P6EXIT	17	27066.208
P2EXIT	14	30983.339	P7EXIT	17	28288.033
P3EXIT	11	30189.912	P8EXIT	13	30744.843
P4EXIT	16	27337.809	P9EXIT	0	0
P5EXIT	18	27933.418	P10EXIT	16	29664.405

The results are almost identical to those of the ASAP model [ZHA05]. However, the simulation time of the CSPE model is significantly slower than that of the ASAP model. For the same model simulating 30 days in real life, the ASAP model took 1~2 minutes to finish, whereas the CSPE model took 2 hours to complete. The reason is mainly due to the introduction of too many dummy workstations in the CSPE model. In the experiments, only 179 of the 1266 simulation objects created are real workstations. A lot of unnecessary simulation events (dummy events) are scheduled every time entities are transferred to the dummy workstations. This incurs a penalty in performance for the CSPE model execution.

7.2.3.2 Integration of Wafer Fab Models

To evaluate the interoperation of CSPs, a borderless fab model is developed by integrating two separate wafer fabrication models. The borderless fab concept was introduced to share capacity

across different wafer fabrication plants in close proximity so as to maximize the equipment utilization level. Lots will be rerouted from one fab to another when the utilization level at the source fab is high or when a particular piece of equipment breaks down. Wafer lots are moved from one fab to another at bottleneck production steps, balancing the trade-off in cycle time and transfer costs. However, the fabs may not like to share their process flow information with other fabs. Information hiding can be achieved using distributed simulation where each simulation model is running its own fab. Communication between models through a communication network is only necessary when wafer lot transfer takes place.

The integration of the two wafer fabs is based on the specification for each model as listed below:

- Each fab is producing ten different product types, with a pre-defined arrival rate (both fabs using different arrival rates)
- Each product type follows its own process flow, which typically is around 200-300 steps
- A total of 70 different toolsets are modeled, ranging from 1 to 20 tools within a toolset.
- Each toolset has a different downtime behavior which is defined by the mean-time-between-failure and mean-time-to-repair.
- Production lots are moved from the first fab to the second fab at a bottleneck step, and vice versa
- The Type I IRM is assumed, which means that lots can be moved from one fab to another without buffering constraints

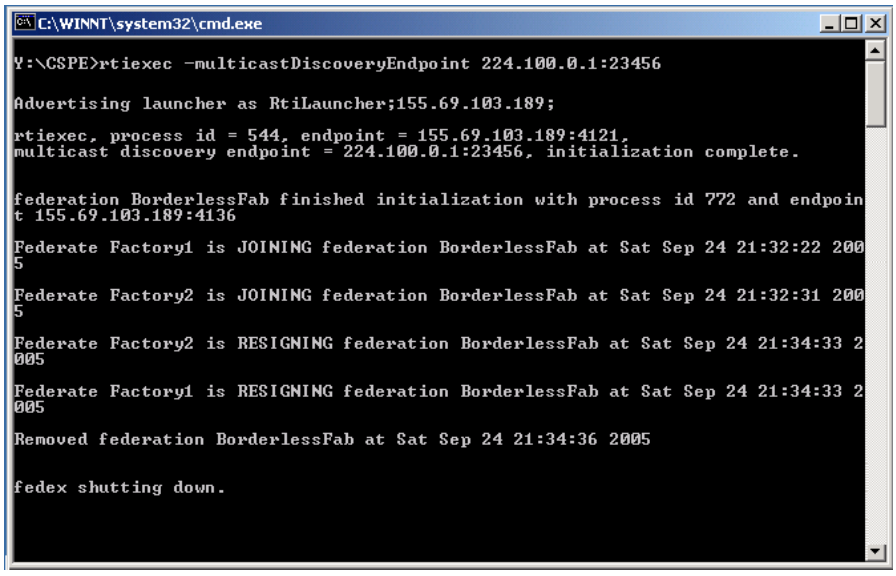


Figure 7.2.4: RTIexec Run

Experiments are conducted on the integrated model, consisting of two wafer fab models. The two models are executed using two CSPE federates, “Factory1” and “Factory2”, and comprise a

federation named "BorderlessFab". Figure 7.2.4 provides a snapshot of the federation information in the rtiexec command prompt window.

Using the generic interface developed and tested based on the CSPE, the ASAP is also extended to support interoperability. As outlined in [GLT05], ASAP fulfils the following requirements that are mandatory for interoperation:

(R1) Ability to initialize the distributed simulation prior to simulation execution
(R2) Ability to suspend the simulation execution
(R3) Access to the time of the next event to be simulated
(R4) Ability to introduce new events/entities from the external source into the event list
(R5) Access to information of simulation objects/entities that are shared among federates

The borderless fab model that comprises of two wafer fabs is also built using ASAP [GLT05]. The two wafer fabs have similar capabilities of producing ten wafer product types but with different capacities. The process flows of the wafer products considered range from 200 to 300 steps. A total of 73 workstation types were modeled in each fab, including the downtime behavior of each workstation. The experiments show the successful interoperability of the ASAP models using the generic interface.

Since ASAP is also extended through the DSManager of the generic architecture, this makes it possible to interoperate the CSPE and ASAP models without too much effort as both of them adopt the same standard. They use a common initialization method, time synchronization algorithm, and data exchange protocol. The only issue that needs to be resolved is the semantics of the data being communicated. This will be further discussed in future work.

7.2.4 Summary

This case study is carried out to investigate the issues and problems of interoperating CSPs in the manufacturing area. Semiconductor wafer manufacturing process flow features are simulated using the available but limited CSPE functionalities. Problems including setting processing times, batching, and linking simulation objects in the CSPE model are solved. In addition to comparing the simulation results of the standalone model running on the CSPE and ASAP, distributed simulation is also developed for a borderless fab model by integrating two separate wafer fabrication models. The experiments have demonstrated successful interoperability between two CSPE models and also two ASAP models with the generic architecture.

The case study also meets the objective of contributing the work of the CSPI-PDG. A practical Type I IRM is investigated and built using the CSPE, verifying the correctness of the generic interface and the capability of the CSPE to support the creation of a practical manufacturing model in both standalone and distributed simulations. Furthermore, the generic interface between the CSP and the HLA RTI can actually be tailored to a real CSP, such as ASAP, to achieve interoperability.

7.3 Integration of a CSP with a Business Application

7.3.1 Motivation

Simulation can be an effective way to evaluate alternative what-if scenarios prior to implementing the best configuration back into the real system. Traditionally, both the business process (representing planning, order management etc.) and operational process (representing shopfloor/warehouse operations etc.) are studied and captured first as simulation models [LJC03], which is a lengthy process and incurs unnecessary cost to translate the business process into a simulation model and vice versa. Moreover, commonly some fidelity of the business logic has to be sacrificed to make the simulation models as simple as possible. A solution is to keep the business application itself and integrate it with a simulation model representing the operational process. It is an example of a symbiotic simulation system [FLP02], which consists of a simulation model interacting with the physical system in a mutually beneficial way. The physical system benefits from the optimized performance that is obtained from the analysis of simulation experiments; and the simulation system benefits from the continuous supply of the latest data and the automatic validation of its simulation outputs. Since currently the simulation models are commonly developed using some kind of CSP, it drives the demand for the integration of CSPs with the physical system / business application. For this purpose, a case study in the aircraft industry area is studied and analyzed.

In the aircraft industry area, the availability of spare components is an important issue in order to ensure the safety and punctuality of flights. It is ideal to have an adequate but not excessive number of spare components in each airport. For this purpose, a business application is designed to manage the inventory level in the airports. The business application is able to move excess inventory of spare component(s) from one airport to another, responding to requests made by aircrafts through the Maintenance Control Centre (MCC) during the flights, or simply for re-balancing the number of components in each airport. To investigate the optimal amount of spare components under different cases/scenarios and algorithms, using the real system is not realizable as it is not very convenient and flexible, and will affect the current working system, and this raises the need for simulation.

As discussed previously, the conventional method of solving this problem is to translate the processes in the business application into a simulation model, using a CSP tool. However this would require a cumbersome translation of one computer representation into another. Any changes to the business application / algorithm mean re-translating the changes into the CSP model and re-validating the model. Therefore, in the case study, the business application itself will be applied directly and will be integrated with a CSPE model representing an operational process, in a distributed simulation environment based on the HLA, utilizing the generic architecture.

7.3.2 Model Design

7.3.2.1 Overview

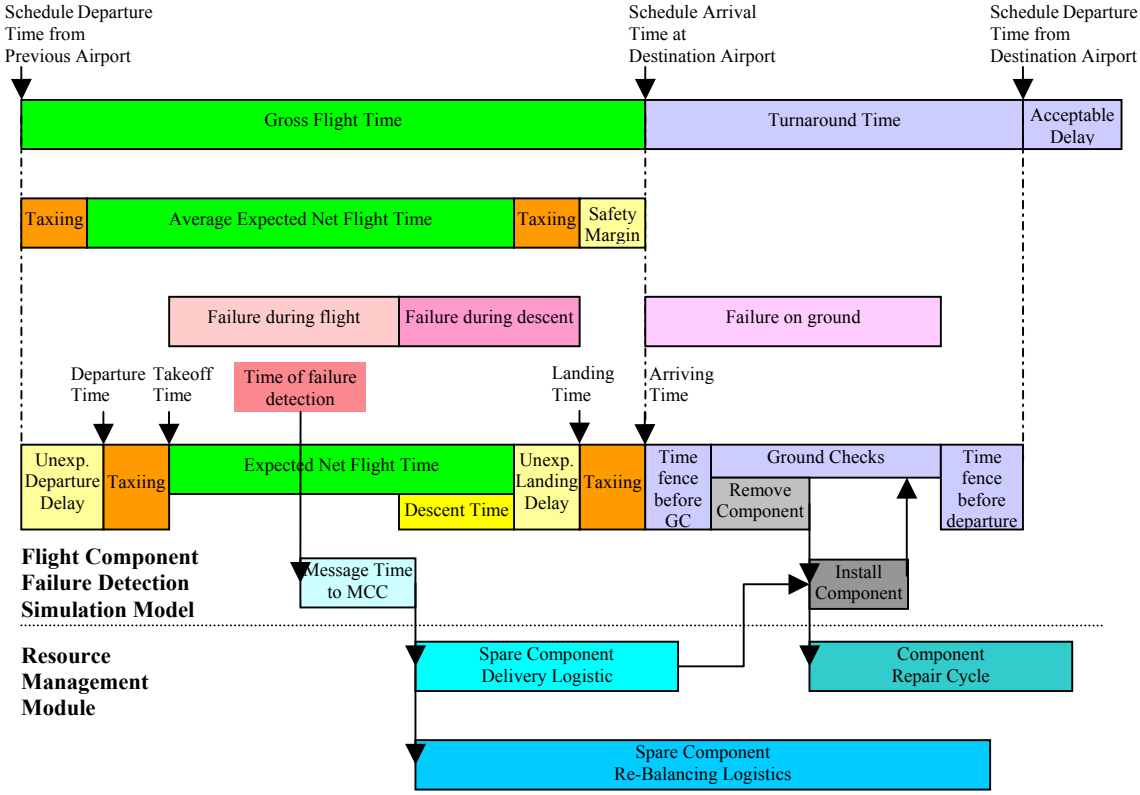


Figure 7.3.1: Aircraft operation cycle

The aircraft industry case study is designed to investigate the possibility of integrating a CSP and a business application. Figure 7.3.1 shows the aircraft operation cycle. It is divided into two major parts: the flight component failure detection simulation model and the resource management module. In the flight failure detection module, various phases of the flight are simulated. This mainly consists of time spent in the air (flying) and time spent on the ground. The total time spent in the air is represented as gross flight time, while the total time spent on the ground is represented as turnaround

time. The acceptable delay is introduced here to provide some extra time to perform the component replacement should a failure happen to an aircraft component. The aircraft component(s) can fail anytime during the flight cycle. A system called Maintenance Control Center (MCC) enables an aircraft to notify the failure before it arrives at the destination airport, thus giving the system more time to provide a spare component at the destination airport. The faulty component will be replaced at the destination airport. A business application (resource management module) is developed to manage the inventory level of the spare components in each airport. It has the ability to move excess spare components from one airport to the other, responding to requests made by the aircraft through the MCC, or simply for re-balancing the number of components in each airport.

7.3.2.2 Design of the Integration Module

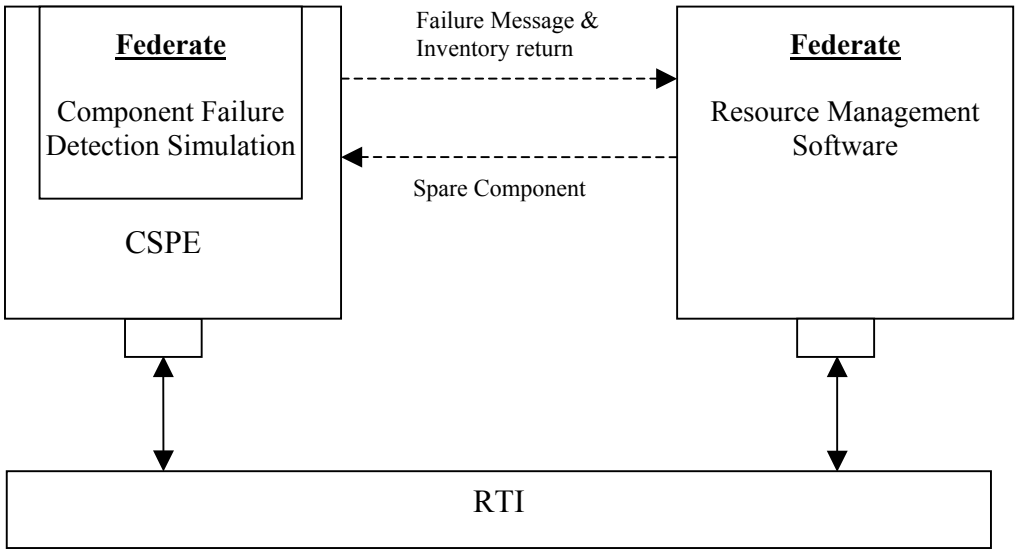


Figure 7.3.2: Overall architecture for the integration module

The overall simulation is built by integrating the flight failure detection module and the resource management module. As shown in Figure 7.3.2, the federation is composed of two federates: one is the business application (resource management module) which is developed by Singapore Institute of Manufacturing Technology (SIMTech), and the other is the simulation model (component failure detection model) built using the CSPE. The information transferred between the business application and the CSPE model is represented as an entity with attributes and exchanged via the underlying HLA RTI. There are two types of entity transferred from the flight component failure detection model. One is for a failure, and the other is for returning a faulty component to the warehouse in the resource management module. There is one type of entity transferred from the resource management module to the first component failure module indicating replacement of a component for a particular

flight. The exchanged entity is tagged with an attribute to indicate that the spare component is exclusively prepared for a particular flight.

7.3.3 Experiments and Discussions

To investigate the optimal inventory levels under different cases/scenarios and algorithms, eight sets of experiments are carried out. In addition to creating the distributed simulation by integrating the model on the CSPE and the business application, similar experiments are also conducted for a standalone simulation using ProModel, one popular CSP in the industry field. The experimental results are compared between the CSPE and ProModel, to validate the integration of the CSPE and the business application based on the proposed generic architecture.

7.3.3.1 Integration of the CSPE and the Business Application

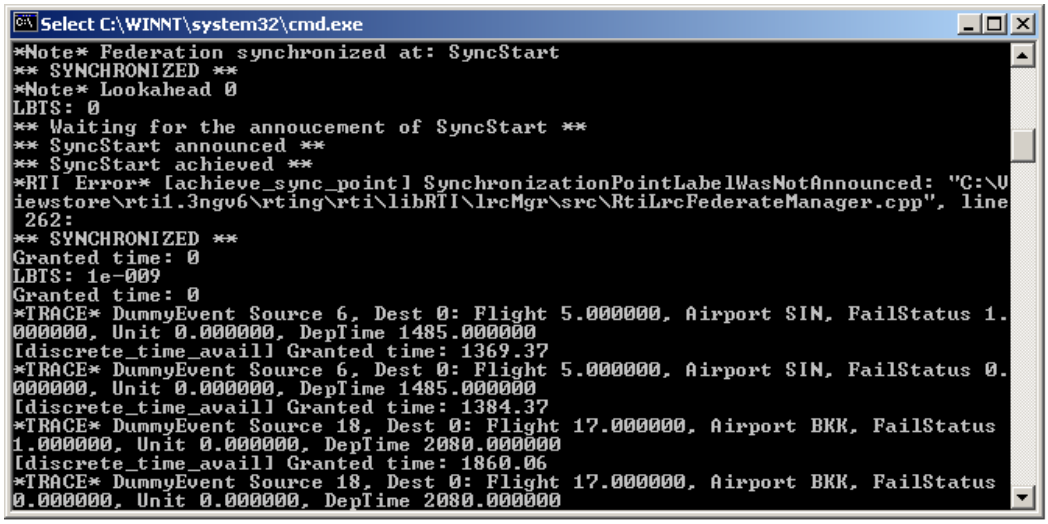


Figure 7.3.3: Inventory management module run

Figure 7.3.3 shows the execution of the inventory management module (with the federate name of bizlogic). The events/messages (entities with attributes) received from the flight component failure detection simulation are shown on the screen. On the other side, the flight component failure detection simulation (with the federate name of fypdemo) is running on the CSPE and some of its activities are monitored using the CSPE GUI as seen in Figure 7.3.4. While the “Received Entities” field displays the number of entities received from the external model (bizlogic federate), the “Transferred Entities” field shows the number of entities sent to the external model (bizlogic federate). These demonstrate the successful integration of the simulation model and the business process.

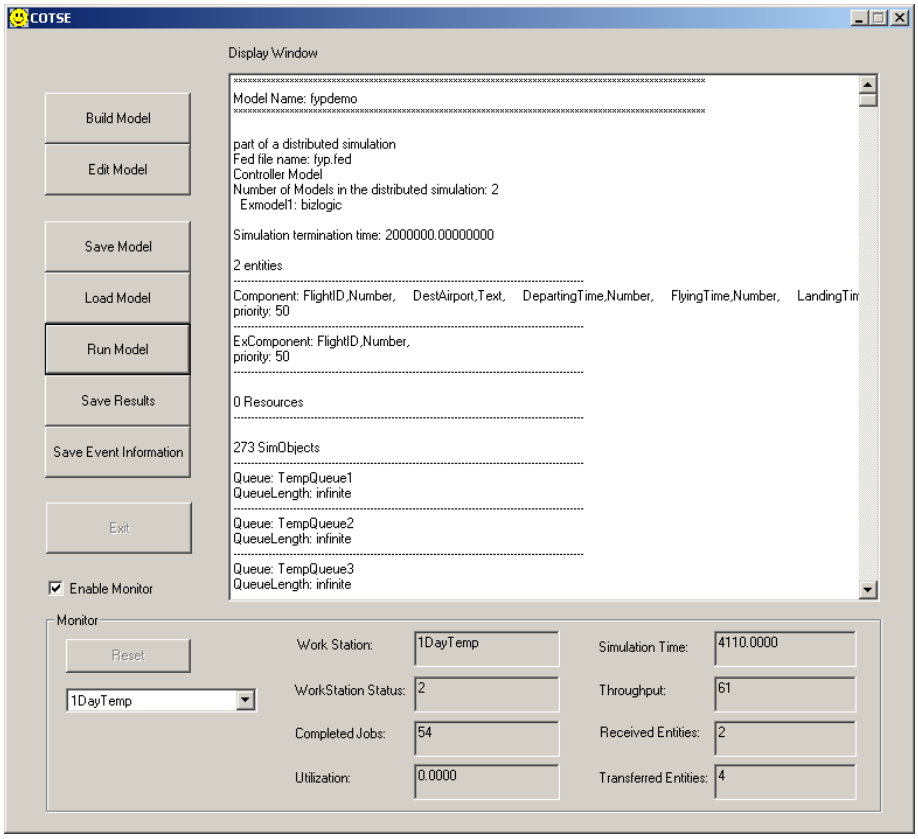


Figure 7.3.4: Failure detection module (CSPE) run

7.3.3.2 Scenarios and Experimental Results

Table 7.3.1: Descriptions of eight scenarios

Scenario	Description
1	MCC activated; Failure detection activated only on the ground; 1 component in each flight.
2	MCC activated; Failure detection activated during descent and on the ground; 1 component in each flight.
3	MCC activated; Failure detection activated during flight, during descent, and on the ground; 1 component in each flight.
4	MCC not activated; Failure detection activated only on the ground; 1 component in each flight.
5	MCC activated; Failure detection activated only on the ground; 2 components in each flight.
6	MCC activated; Failure detection activated during descent and on the ground; 2 components in each flight.
7	MCC activated; Failure detection activated during flight, during descent, and on the ground; 2 components in each flight.
8	MCC not activated; Failure detection activated only on the ground; 2 components in each flight.

Table 7.3.1 shows the descriptions of the eight different scenarios. They are classified mainly by whether the MCC is activated or not, where the failure is detected and the number of components for each flight. The failure can happen any time during the operational cycle. With MCC, the failure can be detected and reported throughout the three phases of the flight: during flight, during descent, and

on the ground. Without MCC, failure can only be detected on the ground. In this model, the failure detection can also be turned on or off for the individual phases of the flight according to the scenario given. The scenarios also demonstrate the capabilities of the model to incorporate either single or multiple components in each flight. While the first four scenarios are models with a single component for each flight, scenarios 5-8 are models with 2 components for each flight. There are 24 flights simulated with 4 destination airports to be observed for all the scenarios. The simulation end time is set as 2,000,000 minutes (about 4 years in minutes).

In Table 7.3.2, the experimental results are compared between the CSPE and ProModel for the eight scenarios. The results show the percentage of flights with faults which can still meet the schedule by being successfully replaced with components. This value is represents as p and is calculated using the following formula:

$$p = \frac{n - f}{n}$$

where n is defined as the number of flights with faults, and f is defined as the number of flights with faults which fail to meet schedule.

Table 7.3.2: Comparison of results for eight scenarios between CSPE and ProModel

Scenario	CSPE	ProModel
1	53.07 %	54.96 %
2	55.48 %	58.80 %
3	71.48 %	76.98 %
4	55.19 %	63.44 %
5	27.00 %	28.48 %
6	31.24 %	30.43 %
7	45.83 %	55.28 %
8	27.93 %	35.89 %

Comparing the results of scenario 3 and other scenarios with a single component (scenarios 1, 2, and 4), and also the results of scenario 7 and other scenarios with two components (scenarios 5, 6, and 8), it can be seen that the performance of the system with MCC activated throughout the whole flight cycle is much better in terms of providing the necessary spare components in time for the aircraft. The longer the failure detection is activated, the better the performance of the system. There is similarity between scenarios 1 and 4, and also between scenarios 5 and 8, where these have only failure detection on the ground.

The experimental results also show that the CSPE and ProModel generate similar trends in terms of the performance of the systems, but giving slightly different values. The difference is due to the fact

that the models are not identical since some assumptions and simplifications are made in the design of the CSPE model because of the limitations of the CSPE. For instance, the CSPE model does not incorporate a 'real' flight frequency/schedule, but assumes the flight happens daily or every 2 days, while the ProModel model does (a 'real' flight frequency might be 1 or 2 times per week, or every day except Monday and Tuesday). That means there are more flights scheduled which will introduce more faults. Since the inventory is limited and the faulty components need some time to be repaired, the value of p will be lower for the CSPE model in Table 7.3.2. Other reasons, such as the different ways to generate random numbers, also lead to differences in the experimental results.

7.3.4 Summary

To investigate the possibility of integrating a CSP with business application, a case study of the aircraft industry is analyzed. A simulation model representing flights around the world is designed and built on the CSPE. The model simulates various phases of the flight cycle and component failures. Component failure can be detected during different phases of the flight cycle and reported to the MCC. Defective components will be replaced with spare components upon arriving at the destination airport. Meanwhile a business application is developed to manage the inventories of spare components in the airports and to simulate the logistic flights carrying the components. The simulation model and the business application are integrated, and the results of the simulations are validated against a similar standalone model created using ProModel.

The experiments show the successful integration of the CSPE and a business process by using the DSManager. Validation of the integration of the CSPE and a business process shows that the simulation results are only slightly different from those of the model developed with ProModel.

7.4 Integration of IBM's WBI Modeler with the HLA

7.4.1 Motivation

IBM's WBI Modeler is one of the products from the IBM's Websphere Business Integration (WBI) suite. It enables a business analyst to design, simulate and document the processes of a company. In this case study, the use of distributed simulation as a way of enhancing performance, reuse and interoperability of WBI Modeler process models is explored. Distributed simulation can facilitate the creation of a large business process consisting of various business components, possibly distributed geographically. Some business components may even be implemented in other commercial packages.

The advent of the High Level Architecture (HLA) standard makes it possible to connect distributed model components together. Among several HLA Runtime Infrastructure (RTI) implementations, the Federation Development Kit (FDK) [FDK] is chosen for this project. FDK is available as freeware from the Georgia Institute of Technology.

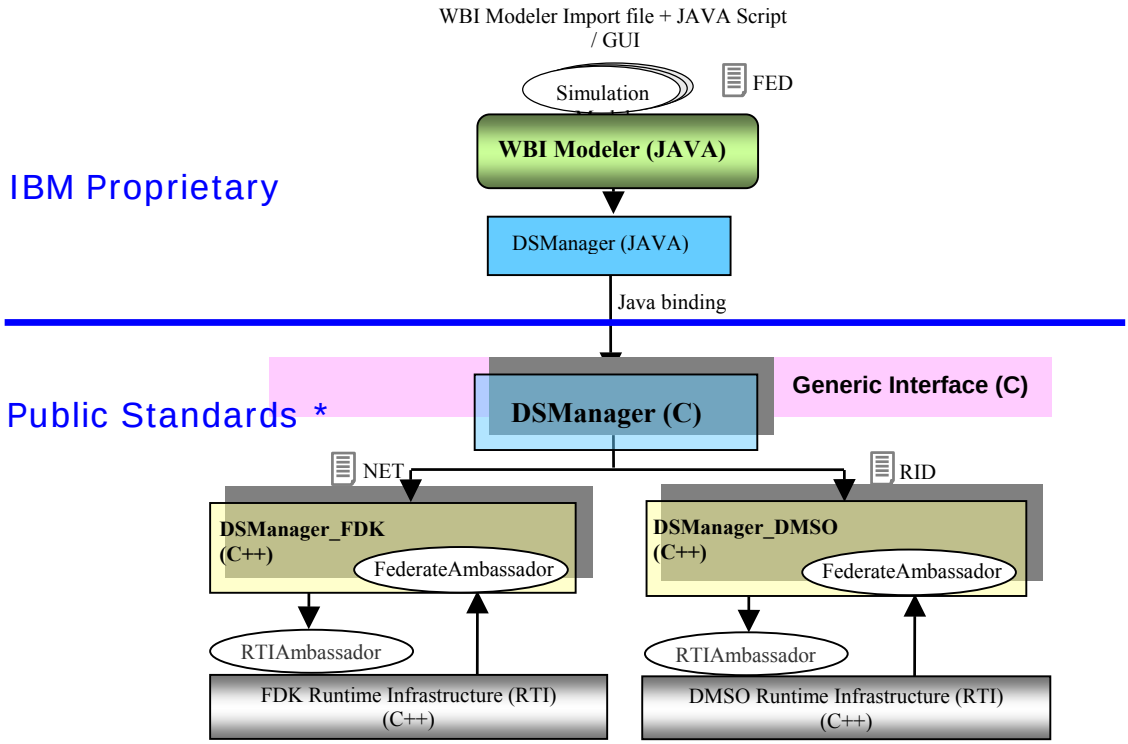


Figure 7.4.1: General framework for integrating the WBI Modeler with the HLA

Figure 7.4.1 shows the general integration architecture. The DSManager is used to integrate the WBI Modeler with the FDK RTI. Currently, the DSManager can interface to either the FDK RTI or the DMSO RTI based on the concept of entity transfer which is easy to understand and invoke. There are minor differences existing between the two implementations of the HLA RTI. For example, while the DMSO RTI uses a .rid file to describe the network topology, the FDK RTI requires a .net file instead. Necessary files should be provided corresponding to the selected RTI implementation. A Java version of FDK RTI is not available but the WBI Modeler is written using Java. To solve this problem, a Java binding of DSManager has been developed for Java-based simulation packages such as WBI Modeler. With proprietary access to the source code and APIs of the WBI Modeler, experimental modifications have been made to invoke corresponding methods in the DSManager. Process analysts can design model components in a “plug & play” manner without worrying about interoperability.

7.4.2 Modification to WBI Modeler

7.4.2.1 Modification to User Interface

The integration of the WBI Modeler and the DSManager is at the API level of WBI Modeler as it only involves the pure simulation engine with an import file describing the model. The modification of the user interface is therefore made in the import file instead of the GUI. The import file is similar to a script file. It allows the user to define all the simulation components as well as their properties. In WBI Modeler, the system to be simulated is a process which consists of nodes (tasks) and directed paths between them (connections). A process can be hierarchical in that any node may refer to another process. Therefore, the main components (as least with the focus on interoperability) are process, task, connection, and port (a connection endpoint). For each type of component, new properties are added to allow the user to declare the interoperability information which is needed by the WBI Modeler simulation engine. For instance, the top-level process supports the properties of 'joinDS' (whether the model needs to join a distributed simulation), 'fedFileName' (the name of the FED file), 'time-constrained' and 'time-regulating' (time policy). Besides, a task could have the property 'import' (similar as an external entry point) or 'export' (similar as an external exit point). In this way, the necessary information is defined in the import file and will be interpreted by the WBI Modeler simulation engine to integrate with the DSManager.

7.4.2.2 Modification to Simulation Engine

As discussed in previous chapters, the CSP needs to invoke corresponding methods provided by the DSManager to support distributed simulation. In addition to facilitate model execution, the CSP simulation engine should also transfer all the necessary interoperability information to the DSManager and receive the returned information from the underlying HLA RTI. The modification is very similar to the work required to link the CSPE with the DSManager. In the initialization phase, the WBI Modeler registers its model as part of a distributed simulation if it is not a standalone model. Then it needs to forward necessary interoperability information of the distributed simulation to the DSManager, such as the name of the local model, the name of the distributed simulation, the name of the FED/FDD file, time policy, and lookahead. It also needs to tell the DSManager what entities its model will exchange with other models in the distributed simulation. During the simulation execution, the WBI Modeler needs to advance the simulation time by providing the time of its next event. As a result of time advancement, it may receive entities with attributes sent from other models. The WBI Modeler then searches the appropriate external entry point from which each entity will be received and schedules a new entity arrival event with associated timestamp in the future event list.

Conversely, if any entity leaves the local model via an external exit point, the WBI Modeler will send it to the receiving model by invoking corresponding functions of the DSManager. When some terminating condition is met, e.g., the simulation end time, the WBI Modeler will inform the DSManager that its model has finished its own simulation tasks and wishes to leave the distributed simulation. After receiving a reply from the DSManager, the WBI Modeler can finally terminate the simulation execution of its local model.

7.4.3 Evaluation

7.4.3.1 Demand Conditioning Business Process Model

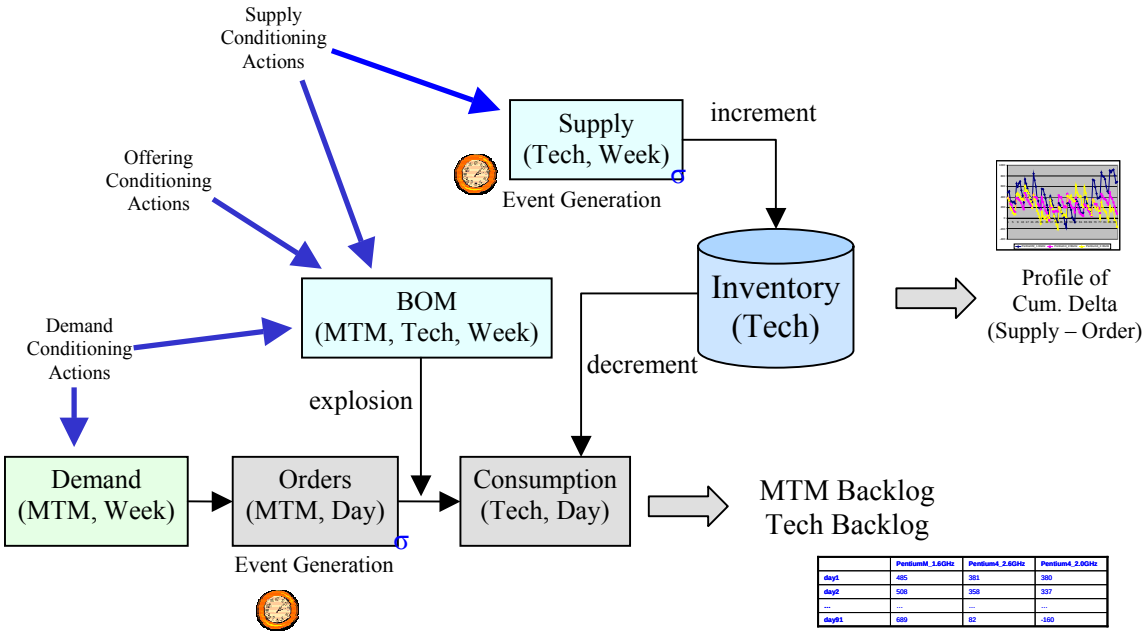


Figure 7.4.2: Overview of IBM demand conditioning process simulation

The IBM Demand Conditioning process [HUL03] is chosen as a case study for the HLA-enabled WBI Modeler. As shown in Figure 7.4.2, the IBM Demand Conditioning process is triggered by an imbalance (not shown in Figure 7.4.2) between supply and demand of computer components to assemble the machine. Based on a known imbalance, decision makers have generated three different types of potential actions to resolve the imbalance: supply conditioning, demand conditioning and offering conditioning. Supply conditioning focuses on cooperating with suppliers to improve flexibility in supply to react to customer demand. Demand conditioning focuses on providing a dynamic sales plan (pricing action, promotion, etc.) which can be changed in reaction to the supply imbalance. Offering conditioning focuses on identifying alternative products for the customers to

resolve the supply imbalances. The simulation must test these potential actions to see how effective they are in reducing the imbalance. In the distributed simulation as shown in Figure 7.4.3, this process has been divided into two component models, where model B includes the BOM (Bill of Materials) and Consumption steps and model A includes all the other steps shown in Figure 7.4.2. As can be seen in Figure 7.4.3, "ExSupplyExit", "ExOrderExit", "ExSupplyEntry" and "ExOrderEntry" are special tasks modified to support distributed simulation, which are similar to the external entry points and external exit points in the CSPE.

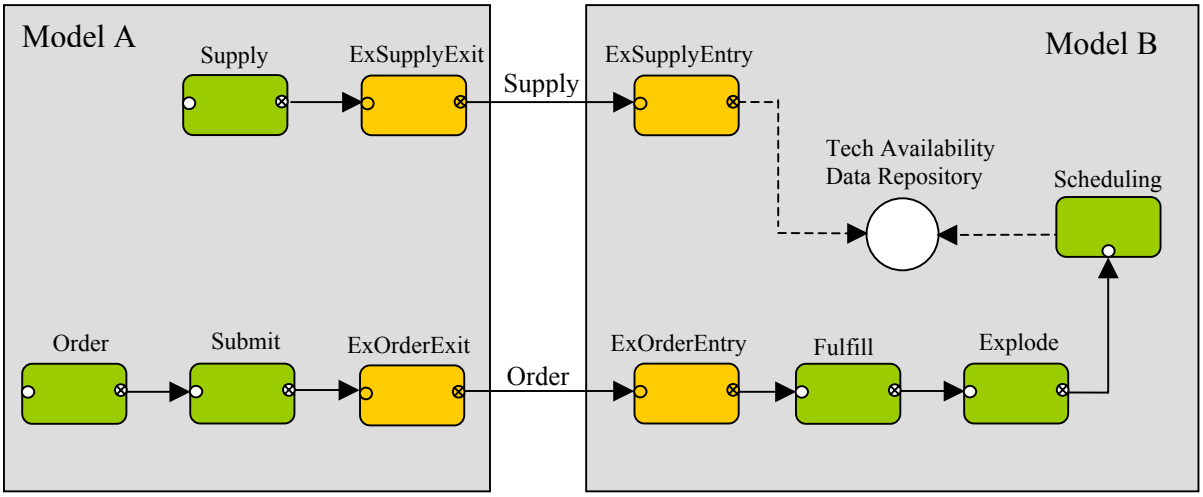


Figure 7.4.3: Distributed model for IBM demand conditioning process

Experiments are conducted with two objectives. The first objective is to evaluate the correctness of the modified WBI Modeler. The same process is run in both standalone and distributed simulations. The inventory is collected each day for experimental results. Both runs generated the identical results when using the same random seeds. The second objective is to compare the runtime performance between the standalone and distributed simulations. In this set of experiments, two experimental factors are varied. One factor is the event granularity, which is defined as the computation time taken to process a task. This factor allows us to vary the computation time to reflect the processing load of the simulation over multiple computers against a sequential simulation. The other factor is lookahead, which is very important for conservative synchronization in distributed simulation. Here lookahead is calculated using the smallest processing time of all tasks prior to the exit points in the model. In the future, algorithms may be investigated to achieve a larger lookahead.

7.4.3.2 Performance Results and Analysis

Figure 7.4.3 shows the execution time versus the event granularity per task. The performance of the standalone simulation is compared to that of three distributed simulations with different lookahead values. The performance of the standalone simulation is better than the distributed simulation when

the event granularity is low, because of the communication overhead in the distributed simulation. When event granularity increases, the distributed simulations (especially with large lookahead) outperform the standalone simulation. This improvement can be attributed to the fact that parallelism is exploited, since the computation with larger time can be conducted concurrently.

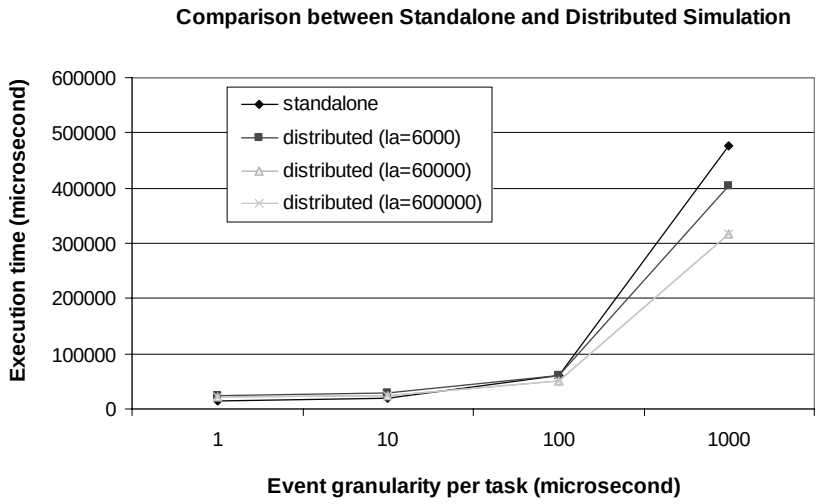


Figure 7.4.4: Comparison of performance between standalone and distributed simulation

7.4.4 Summary

From the above experiments, it can be seen that the HLA-enabled WBI Modeler can support both standalone and distributed simulations and generate correct simulation results. For some kinds of models, distributed simulation can improve runtime performance or facilitate the creation of those models which are too large to execute on a single computer. It is relatively easy to add HLA capabilities to the WBI Modeler, although it requires access to proprietary source code and APIs. The enhancement of the WBI Modeler provides a good example for other real CSPs to support distributed simulation based on the proposed generic architecture in this work.

CHAPTER

8

CONCLUSIONS AND FUTURE WORK

8.1 Achievements

This research is driven by the demand to create large-scale distributed simulations composed of multiple simulation components, even geographically dispersed. In recent years CSPs have become popular as a means of developing simulation models with reduced risk and cost while increasing the functionality and capability of the system. However, different CSPs have diverse characteristics and different degrees of extensibility, which makes the integration of the CSPs difficult. Besides, the complexity of the HLA standard itself and the lack of expertise in distributed simulation (e.g. time synchronization algorithms) also become a roadblock to widespread adoption of the HLA in linking CSPs. A standards-based approach to the integration of the CSPs is required, and can make it possible to build distributed simulations composed of CSP-based simulation components.

After reviewing existing work for integrating CSPs with the HLA and possible implementation approaches, a generic architecture is proposed which adopts a nearly implicit approach from the modeler's point of view. Using this most user-friendly way, the modelers only need to focus on the model design without worrying about the communication with the HLA RTI from their simulation model. This is achieved by the cooperation between the simulation model, the CSP, and a middleware made up of two layers: the DSManager and the extended HLA RTI (RTI+). The DSManager hides the HLA concept from the CSP and the modeler, and wraps the two-part communication with the RTI+ into a one-direction interface to the CSP. The interface provides a set of C functions supporting entity transfer, which are easy to understand and invoke from the CSP. As another part of the middleware, the RTI+ provides the possibility to use alternative time management algorithms in a middleware approach, offering more efficient execution.

As discussed previously, the currently available CSPs are heterogeneous in terms of their properties and degree of extensibility, which makes it extremely difficult to apply the generic architecture to real CSPs in a standardized approach. The CSP Emulator (CSPE) is developed to investigate the requirements and solutions for the integration of CSPs and the HLA. The CSPE emulates the functionality of real CSPs and supports the creation of standalone simulations through a simple GUI or a specific data file. Its simulation engine adopts the three phase approach to manage the activities of the entities and trace the life history of each entity in a simulation. By invoking the interface provided by the DSManager, the CSPE also allows the modeler to build and link component models in a distributed simulation. Currently, the CSPE can be linked with the DSManager designed for asynchronous entity passing (CSPI-PDG Type I IRM) and synchronous entity passing (Type II IRM). The correctness of the CSPE and the generic interface is evaluated for typical CSPI-PDG Type I and Type II IRMs using both deterministic and stochastic models, and simulation results are compared between the CSPE and Simul8, one of the popular CSPs. Based on the work of the integration of the CSPE with the HLA, the generic interface has also been applied for some real CSPs, Autosched AP (ASAP) and WBI Modeler, to support the interoperability of distributed simulation models. All this work provides a suggestion as to how current CSPs may be HLA enabled to support distributed simulation.

This research also directly contributes to the efforts of an international standards committee, referred to as CSPI-PDG. CSPI-PDG is recognized as an official product development group by the Simulation Interoperability Standards Organization (SISO). This group involves representative researchers, users, and vendors in the standardization process, and is dedicated to creating a standardized approach for the interoperation of CSP-based discrete event models using the IEEE 1516-2000 High Level Architecture. This work, on both the interoperability solutions for Type I and Type II IRMs and the CSP Emulator (CSPE), forms the basis of products to be proposed for standardization by SISO.

In addition to the conservative synchronization approach, the optimistic synchronization approach is also investigated in the generic architecture. For some types of models where lookahead is difficult to exploit and has to set as zero or near zero, optimistic synchronization approaches may gain benefits as compared to conservative synchronization. One example is the Type II bounded buffer IRM where near zero lookahead is introduced by the requirement for updating status information. To relieve the modeler from the burden of handling problems in the optimistic approach, the RTI+ middleware is introduced, including a rollback controller to handle most of the complicated rollback procedure on behalf of the CSP and the simulation model. The simulation modeler can still build their simulation federate as before and link the middleware library without worrying about the underlying implementation details.

For some other situations where zero lookahead exists, performance can also be improved even when using the conservative synchronization approach. One example is shared state which is also a characteristic of Type III and Type V IRMs. The zero lookahead is incurred because the state variable has to be written and read at the same instant of simulation time. Two algorithms are proposed in which some of the TSO messages that cause zero lookahead are replaced with RO messages. This removes the time constraint that these messages impose on the calculation of the lookahead, which in turn will improve the time advancement rate of federates. To free the users from the complex details of the implementation, a state manager is introduced into RTI+ middleware. Meanwhile, the original semantics of RTI APIs still remain for the easy use by the simulation developers.

Based on the proposed generic architecture and the CSPE, some case studies are also conducted and analyzed. One is to develop a semiconductor wafer manufacturing model using the CSPE. By introducing more advanced features to meet the requirements from model analysis, a standalone model is created and evaluated by comparing with a similar model running on ASAP. Experiments are also designed for a borderless fab simulation by linking two wafer fab models running on two CSPEs, showing successful interoperability. Another case study is to integrate a simulation model built using the CSPE with a business application in the aircraft industry area. The CSPE model simulating aircraft component failures on flights communicates with the resource management module (business application) which includes the simulation of logistic flights and the decision making algorithm. The simulation model and the business application are integrated, and the results of the simulations are validated with a similar standalone model created using ProModel, another popular CSP.

In addition to the two case models built using the CSPE, another case study is carried out to enable a real CSP, IBM's WBI Modeler, to support distributed simulation. By proprietary access to the source code and APIs of the WBI Modeler, experimental modifications are made to integrate with the DSManager. The HLA-enabled WBI Modeler is also verified by a set of experiments using a demand conditioning process, in which mismatches between supply and demand are identified and corrective actions are initiated. These case studies show it is meaningful to apply the distributed simulation to some practical simulation models on the CSPE as well as some real CSPs using the proposed generic architecture. The successful implementation as well as the identification of the problems involved contributes to the future standardized approach to the interoperability of other CSPs.

8.2 Future Work

8.2.1 Interoperability of Other CSPI-PDG IRMs

As stated in section 2.4, there are six types of CSPI-PDG IRMs which capture different interoperability problems. At the current stage, the project described in this thesis mostly involves the Type I and Type II IRMs and appropriate solutions have been proposed. Using the CSPE, benchmarks can be conducted conveniently to investigate and to compare various interoperability solutions. As part of a project in the Integrated Manufacturing & Service Systems Programme (IMSS) [LLG05], our future work will focus on the interoperation of CSP simulation models and time synchronization mechanisms for the other four types of IRMs. This will cover shared resources, shared events, shared data structures and shared conveyors. Some investigations have already done for similar problems to these, e.g., shared state which has to be written and read at the same instant of simulation time. As it is known, the major issue in these models is the requirement for zero lookahead. Some algorithms have been proposed to improve the performance by replacing the TSO messages with RO messages to reduce the constraint for time advancement. Alternatively, it is also interesting to see whether optimistic synchronization can gain any benefit in these situations. It is also preferable to develop one general DSManager which can be applied to all these six types of IRMs. Currently, the DSManager designed for type II IRM also supports type I IRM without introducing much overhead. In this way, the CSPs can allow the modelers to build their simulation models without identifying the type of the model itself.

8.2.2 Integration of Other CSPs with the HLA and Data Engineering Issues

The proposed generic architecture is first investigated and verified using benchmark experiments on the CSPE. Based on this work, the generic architecture is further applied to some real CSPs, ASAP and WBI Modeler to support distributed simulation. These HLA-enabled CSPs make good examples and suggestions for future integration of other CSPs.

Currently, this research mainly focuses on the integration of separate simulation model components running on the same CSP or CSPE, e.g., two ASAP models. However, it is possible a large complex simulation system may comprise of a number of submodels, each independently built using a CSP specifically suited for its own purpose. For example, for a 300mm wafer fab manufacturing system, AutoMod is typically used for the material handling simulation, while ASAP is used for the process

flow simulation. Distributed simulation technology is thus required to integrate and interoperate the material handling and manufacturing process models. Through the DSManager of the generic architecture, AutoMod and ASAP can be enhanced to support distributed simulation. They will use a common initialization method, time synchronization algorithm, and data exchange protocol. This will simplify the efforts in putting the two simulation packages together tremendously as both AutoMod and ASAP will adopt the same standard.

The integration of two models built using different CSPs, however, is not a straightforward task. One important issue that needs to be resolved is the semantics of the data being communicated. A problem may arise from the fact that some of the information is represented using different data types or different representations. For example, suppose the borderless fab model discussed in Section 7.2 is built by integrating one wafer fab running on the CSPE with another on ASAP. The CSPE model uses Step ID, a global index on all possible steps, to distinguish individual steps with the same step numbers. In contrast, the ASAP model makes use of available functionalities and uses step numbers directly. This problem cannot be solved by merely adopting a data exchange standard since the meaning of the data is different between the two models. The models are still holding the same information, but in very different ways. A possible solution is to implement a data translation layer at one of the fabs, which maps the data from one representation to another [GTT05]. To truly realize a "plug-and-play" simulation, a more general approach is required to solve this problem. Model-based data engineering [TOD05] is one of the potential solutions that will be investigated to facilitate the translation of dialects of an individual simulation component into a common language, e.g., XML (Extensible Markup Language) specially designed for simulation purposes.

8.2.3 Performance Improvements

Further work is also necessary to improve the performance of distributed simulation. For the conservative synchronization approach, it is essential to exploit the value of lookahead as much as possible. In the current CSPE, only the travel time is calculated as the value of lookahead. In fact, the lookahead value can be further optimized based on the scenario of the model, for example, by using the travel time plus the processing time of the last workstation prior to an exit point. The problem is that the DSManager has no idea of the model structure and cannot calculate the value of the lookahead on behalf of the CSP and the model. The CSP vendors, however, may not be willing to perform the calculation either since it is not a simple task. A compromise solution is to require the CSP to provide some necessary information to the DSManager for the automatic calculation of the lookahead.

For some other types of models, e.g., Type II IRM, the lookahead has to be set as near zero or zero. A better performance may be obtained by adopting an optimistic synchronization approach. In the rollback controller, only the basic rollback procedure is implemented without applying possible optimization algorithms (e.g., some algorithms discussed in Section 2.1.3). Further work can be done in this area. To reduce the cost of the optimistic approach, it is essential to utilize efficient algorithms to optimize the handling of state saving, rollback and fossil collection. The selection of appropriate algorithms is also closely related to the specific properties of the simulation models. However, to apply such application specific algorithms in a middleware approach is not simple, since the particular information of the models is not available to the rollback controller. It may be solved by the cooperation between the CSP and the rollback controller. Also, it would be interesting to carry out performance experiments involving complex models on heterogeneous architectures formed by shared memory, LAN and Internet connected machines, or even in a Grid environment.

8.2.4 An XML-Based Generic Simulation Language

Currently available CSPs are heterogeneous in terms of data representation and semantic interpretation. A model built using a certain CSP cannot easily be loaded and reused by other CSPs. Furthermore, the SOM/FOM development also involves an extensive amount of work, which may need to be repeated when a simulation component is used to form another distributed simulation. Some work has been done to develop a common simulation language based on XML [XML] that is extensible so that individual users or simulation vendors may extend the language to meet their specific requirements. Examples of such work includes XMSF (Extensible Modeling and Simulation Framework) [XMSF], SRML (Simulation Reference Markup Language [SRML][REI02], XPSL (Extensible Pipeline Simulation Language) [XPSL] and PIMODES (Process Interaction Modeling Ontology for Discrete Event Simulation [LAC06].

Future work should summarize and evaluate the existing work, and then propose an XML-based generic simulation language for the interoperability of CSPs. This will be part of a future generic architecture as shown in Figure 8.2.1. In this vision, simulation model components would be written in the generic simulation language which could represent the model using a general-purpose schema. Instead of requiring a standardized simulation engine, it is hoped the model could be loaded and interpreted by the simulation engines provided by current CSPs. Each CSP has its own distinctive simulation engine to support simulation and modeling and it is desirable to add new functionality to interpret the model written in the generic simulation language. Additionally, it allows different implementations of the HLA RTI, such as DMSO RTI, FDK RTI, and pRTI [PRTI] to be used. Ideally, the DSManager is generic enough to link with each RTI implementation transparently to the

CSPs. The CSP only needs to inform the DSManager of the type of the RTI implementation according to the availability of the HLA RTI software. In the future vision, the model could be built and modified easily independent of the platforms and the CSPs. By integrating with the DSManager middleware and supporting HLA RTI, the CSPs are able to talk with each other and form a distributed simulation, thus facilitating reuse and interoperability.

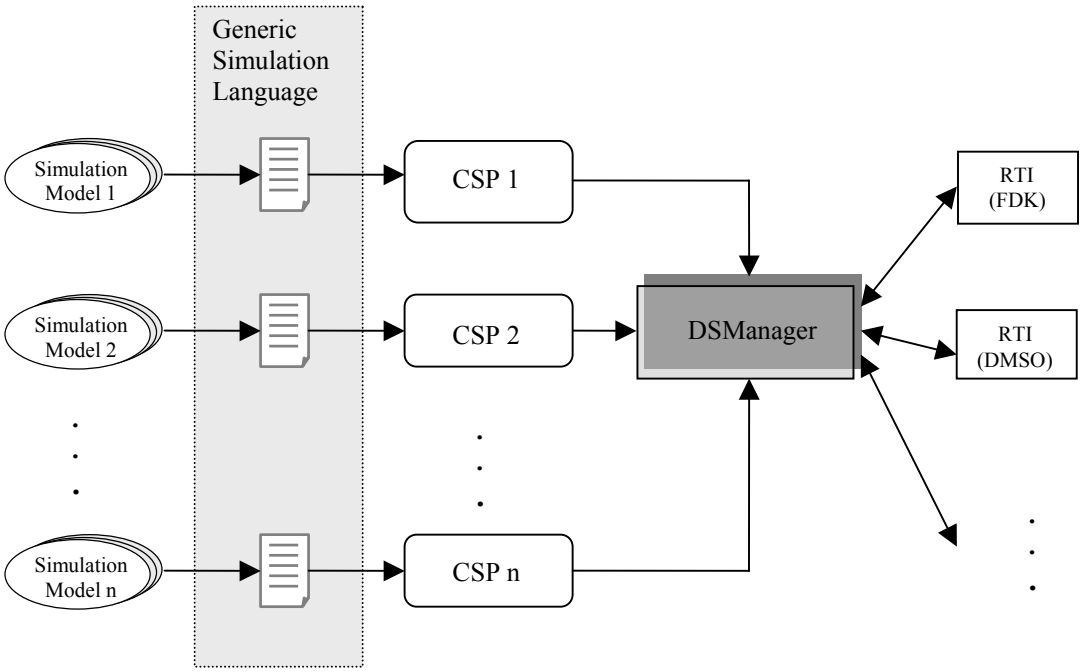


Figure 8.2.1: Future vision

8.3 Summary

To achieve the interoperability of current popular CSPs, a generic architecture is proposed. The DSManager provides a generic interface between the CSP and the HLA RTI which can then be tailored to specific CSPs. The DSManager has been successfully linked with the CSPE as well as some real CSPs, including ASAP and WBI Modeler, enabling these CSPs to be HLA-compatible. Another contribution of this research comes from providing solutions to synchronization issues. These include a rollback controller to handle the rollback procedure in the optimistic synchronization approach and a state manager to relax the constraint introduced by shared state in the conservative synchronization approach. Both of them are implemented in middleware that can release the CSPs as well as the simulation modeler from complex details of the implementation. The project described in this thesis also directly contributes to the efforts of the CSPI-PDG, an international standards committee. In this way, it is hoped more CSPs will benefit from this work to support distributed simulation, thus facilitating reuse and interoperability of CSP components.

APPENDIX A: DSMANAGER REFERENCE MANUAL

This section describes the functions provided by the DSManager. They can be used by the developers to extend available CSPs or their own simulation programs to support the distributed simulation. Additionally, the "internal" HLA functions called by each of the DSManager services are also provided for the users who are interested in the underlying implementation. The functions are divided into four groups: initialization, simulation execution, termination and supporting services. In each groups, the functions are listed in alphabetical order.

A.1 Initialization

A.1.1 DS_registerController()

ABSTRACT
DS_registerController tries to register the controller in the whole distributed simulation.

SYNOPSIS
Procedure DS_registerController (
int noOfModels);
returning int;

ARGUMENTS
noOfModels
This int value specifies the number of simulation models in the whole distributed simulation.

DESCRIPTION
DS_registerController is used to set the controller in the distributed simulation. The controller is in charge of registering synchronization points as well as starting and ending the simulation process. Only one model in the distributed simulation can be registered as the controller.

RETURN VALUES
The return value is –1 if some RTI unexpected errors occurred. Or else 0 is returned for the normal case.

HLA FUNCTIONS INVOKED
None

A.1.2 DS_registerDistributedSimulation()

ABSTRACT

DS_registerDistributedSimulation tries to register a model to join a specified distributed simulation.

SYNOPSIS

```
Procedure DS_registerDistributedSimulation (
        const char* DSName,
        int DSNameSize,
        const char* modelName,
        int modelNameSize,
        const char* fedFileName,
        int fedFileNameSize)
        returning int;

Procedure DS_registerDistributedSimulation (
        const char* DSName,
        int DSNameSize,
        const char* modelName,
        int modelNameSize,
        const char* fedFileName,
        int fedFileNameSize,
        int isTimeRegulating,
        int isTimeConstrained)
        returning int;
```

ARGUMENTS

DSName

This address points to a buffer storing the name of the distributed simulation which the model will join as a part of the simulation.

DSNameSize

This int value specifies the size of the buffer storing the name of the distributed simulation.

ModelName

This address points to a buffer storing the name of the registered model.

ModelNameSize

This int value specifies the size of the buffer storing the name of the registered model.

FedFileName

This address points to a buffer storing the name of the federation execution data (*.fed) file which will be used when the distributed simulation is initialized.

FedFileNameSize

This int value specifies the size of the buffer storing the name of the *.fed file.

IsTimeRegulating

This int value specifies whether the registered model is time-regulating. 1 is time-regulating which means the model will regulate the advancement of the federation time.

IsTimeConstrained

This int value specifies whether the registered model is time-constrained. 1 is time-constrained which means the model's time advancement is constrained by time-regulating models in the same federation.

DESCRIPTION

DS_registerDistributedSimulation is used to register one model to join a specified distributed simulation. Each model constituting the distributed simulation should call this function to be a part of the specified distributed simulation.

RETURN VALUES

The return value is –1 if some RTI unexpected errors occurred. Or else 0 is returned for normal case.

HLA FUNCTIONS INVOKED

createFederationExecution ()
joinFederationExecution ()
registerFederationSynchronizationPoint ()
synchronizationPointAchieved ()
enableAsynchronousDelivery ()
enableTimeConstrained ()
enableTimeRegulation ()
modifyLookahead ()

A.1.3 DS_registerInEntity()

ABSTRACT

DS_registerInEntity tries to register an entity which will be received by the model.

SYNOPSIS

```
Procedure DS_registerInEntity (
        const char* entityName,
        int entityNameSize,
        const char* sourceModelName,
        int sourceModelNameSize,
        bool isRestricted)
        returning int;
```

ARGUMENTS

entityName

This address points to a buffer storing the name of the entity which will be received from other remote models in the distributed simulation.

EntityNameSize

This int value specifies the size of the buffer storing the name of the entity.

SourceModelName

This address points to a buffer storing the name of the remote model from which the registered entity will be received.

SourceModelNameSize

This int value specifies the size of the buffer storing the name of the remote model.

IsRestricted

This bool value specifies whether the external entry to receive the specified entity is restricted. If at least one destination of the entry is a workstation with bounded capacity or a queue with bounded length, the entry is considered as a restricted one. For restricted external entry, status information of the entry needs to be sent by the local model. Correspondingly, the model which sends the entity to this entry needs to check the status information. In this situation, lookahead must be set to near zero.

DESCRIPTION

DS_registerInEntity is used to register the entity which will be received by the model. The entity is sent by those models which register this type of entity as OutEntity. The unique entityID will be returned determined by the specified entityName and sourceModelName. This entityID will be used to receive an entity in the function DS_receiveEntity.

RETURN VALUES

The return value is –1 if some RTI unexpected errors occurred. Or else the positive value is a unique entityID.

HLA FUNCTIONS INVOKED

```
subscribeInteractionClass ()
        // subscribe entity information class
publishInteractionClass ()
        // publish status&priority information class and isRestricted information class
        // (specially for Type II IRM)
```

A.1.4 DS_registerOutEntity()

ABSTRACT

DS_registerOutEntity tries to register the entity which will be sent by the model to other remote models.

SYNOPSIS

```
Procedure DS_registerOutEntity (
        const char* entityName,
        int entityNameSize,
        const char* destinationModelName,
        int destinationModelNameSize)
        returning int;
```

ARGUMENTS

entityName

This address points to a buffer storing the name of the entity which will be sent to other remote models in the distributed simulation.

EntityNameSize

This int value specifies the size of the buffer storing the name of the entity.

DestinationModelName

This address points to a buffer storing the name of the remote model to which the registered entity will be sent.

DestinationModelNameSize

This int value specifies the size of the buffer storing the name of the remote model.

DESCRIPTION

DS_registerOutEntity is used to register the entity which will sent by the model. The entity will be received by those models which register the type of entity as InEntity. The unique entityID will be returned determined by the specified entityName and destinationModelName. This entityID will be used to transfer an entity in the function transferEntity.

RETURN VALUES

The return value is –1 if some RTI unexpected errors occurred. Or else the positive value is a unique entityID.

HLA FUNCTIONS INVOKED

```
publishInteractionClass ()
        // publish entity information class
subscribeInteractionClass ()
        // subscribe status&priority information class and isRestricted information class
        // (specially for Type II IRM)
```

A.1.5 DS_runDistributedSimulation()

ABSTRACT
DS_runDistributedSimulation starts simulation run after all the model components are synchronized.

SYNOPSIS
Procedure DS_runDistributedSimulation ()
returning void;

ARGUMENTS
None.

DESCRIPTION
DS_runDistributedSimulation is invoked after the model reisters distributed simulation as well as declaring the entities it wants to exchange with other models in this distributed simulation. For Type I IRM, this function can be ignored since nothing will be done by the DSManager. However, it is required for Type II IRM since the DSManger needs to send and collect isRestricted information which is used to check whether it is a Type II IRM. If it is a Type II IRM, the lookahead value has to be modified to near zero no matter what value is provided by the model.

RETURN VALUES
None.

HLA FUNCTIONS INVOKED
sendInteractionClass ()
modifyLookahead ()

A.1.6 DS_setLookahead()

ABSTRACT
DS_setLookahead tries to set the lookahead value. If it is not called, the default value is 0.

SYNOPSIS
Procedure DS_setLookahead (
double lookahead)
returning void;

ARGUMENTS
lookahead
This double value specifies the lookahead value. The default value is 0.

DESCRIPTION
DS_setLookahead is used to set the lookahead value. It is important when the conservative synchronization method is chosen. It should be called before the function DS_registerDistributedSimulation is invoked.

RETURN VALUES
None

HLA FUNCTIONS INVOKED
None

A.1.7 DS_setSynMethod()

ABSTRACT

DS_setSynMethod tries to set the synchronization method. If it is not called, the default value is 0 (conservative method).

SYNOPSIS

Procedure DS_setSynMethod (
int method,
double endTime)
returning void;

ARGUMENTS

method

This int value specifies the synchronization method. 0 represents conservative method (as default) and 1 represents optimistic method.

EndTime

This double value specifies the simulation termination time.

DESCRIPTION

DS_setSynMethod is used to set the synchronization method as well as the termination time of the simulation. For the current version, two methods are provided, referred to as the conservative and the optimistic methods.

RETURN VALUES

None

HLA FUNCTIONS INVOKED

None

A.2 Simulation Execution

A.2.1 DS_advanceTime()

ABSTRACT

DS_advanceTime tries to advance simulation time to the time the model requests.

SYNOPSIS

```
Procedure DS_advanceTime (
        double theTime,
        int * nEntityReceived)
returning double;
```

ARGUMENTS

theTime

This double value specifies the simulation time to which the model wishes to advance.

NentityReceived

This pointer to int is provided for the model to receive the number of enitities received during the process of time advancement.

DESCRIPTION

DS_advanceTime is used for the model to advance its simulation time. Usually the time is the time of the next event in the model. The model can get two values from the function. One is the simulation time granted. The other is the number of entities received from other models during the process of advancing time. Then the model can advance its simulation time to the grant time and invoke DS_receiveEntity until all the *nEntityReceived* entities are received.

RETURN VALUES

The return value is the time to which the model has been granted to advance, that is, the new value of the model's simulation time. The value can be equal to or less than the time specified in the parameter theTime. The return value is –1 if some RTI unexpected errors occurred.

HLA FUNCTIONS INVOKED

nextEventRequestAvailable ()

A.2.2 DS_getAttributeValue()

ABSTRACT

DS_getAttributeValue tries to get the value of one specific attribute of one specific entityID.

SYNOPSIS

```
Procedure DS_getAttributeValue (
        int entityID,
        const char* AttriName,
        int AttriNameSize,
        int * valueSize)
returning char*;
```

ARGUMENTS

entityID

This int value specifies the entityID to which the attribute belongs. It can be obtained when invoking DS_registerOutEntity. Or it can also be retrieved by calling the supporting service DS_getOutEntityID.

AttriName

This address points to a buffer storing the name of the attribute for which the value will be got.

AttriNameSize

This int value specifies the size of the buffer storing the name of the attribute.

ValueSize

This pointer to int used for the model to retrieve the size of the buffer which is used to store the value of the specified attribute.

DESCRIPTION

DS_getAttributeValue is used to get the value of one specific attribute of one specific entityID. If there is entity received with the specified entityID after calling DS_receiveEntity, DS_getAttributeValue is invoked to get the value of each attribute subscribed for the specified entityID. If there is more than one entity of the same type (entityID) received at the same time, the DSManager is responsible for forwarding all the values one by one each time the DS_getAttributeValue is invoked.

RETURN VALUES

The return string value is a buffer which stores the value of the specified attribute.

HLA FUNCTIONS INVOKED

None.

A.2.3 DS_getExEntryStatus()

ABSTRACT

DS_getExEntryStatus tries to get the status of the external entry point which is used to receive the specified entity.

SYNOPSIS

Procedure DS_getExEntryStatus (
int entityID)
returning int;

ARGUMENTS

entityID

This int value specifies the entityID which is used to check whether a specific entity can be sent to a specific remote model. It can be obtained when the DS_registerOutEntity is invoked. It can also be retrieved using the supporting service DS_getOutEntityID by providing the entity name as well as the destination model name.

DESCRIPTION

Before sending an entity to the destination model, the source model needs to check the status of the external entry in the destination model to receive the entity. Only when the external entry is idle, can the model send the specified entity.

RETURN VALUES

The return value is 0 if the external entry to receive the specified entity is idle; 1 is returned if the external entry is blocked. Or else –1 is returned to indicate the external entry does not exist.

HLA FUNCTIONS INVOKED

None.

A.2.4 DS_receiveEntity()

ABSTRACT

DS_receiveEntity tries to retrieve the number of entities received during the last time advance request for the specified entityID.

SYNOPSIS

```
Procedure DS_receiveEntity (
        int entityID,
        double * time)
returning init;

Procedure DS_receiveEntity (
        int entityID,
        double * time,
        double * travelTime)
returning init;
```

ARGUMENTS

entityID

This int value specifies the entityID which is used to check whether there is the entity with the entityID received during the last time advance request. The entityID can be obtained when invoking DS_registerInEntity. Or it can also be retrieved by calling the supporting service DS_getInEntityID.

Time

The pointer to double is used for the model to retrieve the time when the entity is sent from the other model.

TravelTime

The pointer to double is used for the model to retrieve the travel time for the received entity.

DESCRIPTION

DS_receiveEntity tries to retrieve the number of entities received during last time advance request for the specified entityID. After calling advanceTime, the model will get the number of entities received from all other models. If the number is larger than 0, the model needs to call DS_receiveEntity to check whether there is any entity with this entityID received. If it is, the model will call DS_getAttributeValue to get the value of each attribute of this entity.

RETURN VALUES

The return value is 0 if the specified entityID does not exist or no new entities are received for the specified entityID during the last time advance request. Or else a positive integer value is returned as the number of the new entities received.

HLA FUNCTIONS INVOKED

None.

A.2.5 DS_setAttributeValue()

ABSTRACT

DS_setAttributeValue tries to set the value of one specific attribute of one specific entity which will be sent by the model to other remote models.

SYNOPSIS

```
Procedure DS_setAttributeValue (
        int entityID,
        const char* AttriName,
        int AttriNameSize,
        const char* value,
        int valueSize)
returning void;
```

ARGUMENTS

entityID

This int value specifies the entityID to which the attribute belongs. It can be obtained when invoking DS_registerOutEntity. Or it can also be retrieved by calling the supporting service DS_getOutEntityID.

AttriName

This address points to a buffer storing the name of the attribute for which the value will be set.

AttriNameSize

This int value specifies the size of the buffer storing the name of the attribute.

Value

This address points to a buffer storing the value of the attribute.

VauleSize

This int value specifies the size of the buffer which is used to store the attribute value.

DESCRIPTION

DS_setAttributeValue is used to set the value of one specific attribute of one specific entity. Before calling DS_transferEntity to send the entity, the model needs to set the value of each attribute which has been published for a specific entity. It should be noted that not all the attributes of the entity need to be shared with other models

RETURN VALUES

None.

HLA FUNCTIONS INVOKED

None.

A.2.6 DS_setExEntryStatus()

ABSTRACT

DS_setExEntryStatus tries to set the status of the external entry point which is used to receive the specified entityID. This information will be sent by the DSManager to the destination model.

SYNOPSIS

```
Procedure DS_setExEntryStatus (
        int entityID,
        int status)
returning void;

Procedure DS_setExEntryStatus (
        int entityID,
        int status,
        int priority)
returning void;
```

ARGUMENTS

entityID

This int value specifies the entityID which is used to check whether a specific entityID can be sent to a specific remote model. It can be obtained when the DS_registerInEntity is invoked. It can also be retrieved using the supporting service DS_getInEntityID by providing the entity name as well as the source model name.

Status

This int value specifies the status of the external entry for the specified entityID.

Priority

This int value specifies the priority of the external entry for the specified entityID. In some models, more than one entry point may be linked to the same output simulation object (e.g., workstation, queue). They may have different priority values. The priority value is used for simultaneous events. The smaller the value, the higher the priority.

DESCRIPTION

When the priority or the status of the external entry changes, the model needs to update the status information of the external entry to receive the specified entityID. An external entry may have more than one output simulation object. For different output simulation objects, the entry may have different priorities. The model needs to check which one is the next destination and to update the priority of the next destination if its value changes.

RETURN VALUES

The return value is –1 if the external entry does not exist or some RTI exception happens. Or else 0 is returned for the normal case.

HLA FUNCTIONS INVOKED

None.

A.2.7 DS_transferEntity()

ABSTRACT

DS_transferEntity tries to send an entity with the specified entityID to other models.

SYNOPSIS

```
Procedure DS_transferEntity (
        int entityID,
        double time)
returning int;

Procedure DS_transferEntity (
        int entityID,
        double time,
        double travelTime)
returning int;
```

ARGUMENTS

entityID

This int value specifies the entityID of the entity with attribute values which the model will send to other models. The entityID can be obtained when invoking DS_registerOutEntity. Or it can also be retrieved by calling the supporting service DS_getOutEntityID.

Time

The double value specifies the time when the entity will be sent by the model.

TravelTime

The double value specifies the time for the entity to travel to destination model. This value could be used as a lookahead value in some situations.

DESCRIPTION

DS_transferEntity is used to send the entity with the specified entityID to other models. After processing some internal events, the model may generate some entities with attribute values which need to be sent to other remote models. Before calling DS_transferEntity to send entity, the model needs to set the value of each attribute which has been published for this entity. It should be noted not all the attributes of the entity need to be shared with other models.

RETURN VALUES

The return value is –1 if some RTI unexpected errors occurred. Or else 0 is returned for normal case.

HLA FUNCTIONS INVOKED

sendInteraction ()

A.3 Termination

A.3.1 DS_terminateDistributedSimualtion()

ABSTRACT

DS_terminateDistributedSimulation tries to terminate the simulation execution.

SYNOPSIS

Procedure DS_terminateDistributedSimulation ()
returning int;

ARGUMENTS

None.

DESCRIPTION

DS_terminateDistributedSimulation is used to terminate the simulation execution. The controller in the distributed simulation will synchronize the terminating point to terminate the whole distributed simulation execution.

RETURN VALUES

The return value is –1 if some RTI unexpected errors occurred. Or else 0 is returned for normal case.

HLA FUNCTIONS INVOKED

registerFederationSynchronizationPoint ()
synchronizationPointAchieved ()
resignFederationExecution ()
destroyFederationExecution ()

A.4 Supporting Services

A.4.1 DS_getInEntityID()

ABSTRACT

DS_getInEntityID tries to get the entityID with the specified entity name and the model name from which the entity is received. This entity must have been registered as one that will be received from other models.

SYNOPSIS

```
Procedure DS_ getInEntityID (
        const char* entityName,
        int entityNameSize,
        const char* sourceModelName,
        int sourceModelNameSize)
        returning int;
```

ARGUMENTS

entityName

This address points to a buffer storing the name of the entity which will be received from other remote models in the distributed simulation.

EntityNameSize

This int value specifies the size of the buffer storing the name of the entity.

SourceModelName

This address points to a buffer storing the name of the remote model from which the registered entity will be received.

SourceModelNameSize

This int value specifies the size of the buffer storing the name of the remote model.

DESCRIPTION

DS_getInEntityID is used to get the unique entityID for the specified entity name and the model name from which the entity is received. The entityID for InEntity is used receive the entity when invoking DS_receiveEntity and DS_getAttributeValue.

RETURN VALUES

The return value is –1 if an entity with the specified entity name and source model name does not exist. Or else the positive value is a unique entityID.

HLA FUNCTIONS INVOKED

None.

A.4.2 DS_getInEntityName()

ABSTRACT

DS_getInEntityName tries to get the entity name and the model name from which the entity is received with the specified entityID. This entity must have been registered as one that will be received from other models.

SYNOPSIS

```
Procedure DS_getInEntityName (
        int entityID,
        int* entityNameSize,
        char** sourceModelName,
        int* sourceModelNameSize)
returning char*;
```

ARGUMENTS

entityID

This int value specifies the entityID.

EntityNameSize

This int value specifies the size of the buffer containing the name of the entity.

SourceModelName

This address points to a buffer storing the name of the remote model from which the registered entity will be sent.

SourceModelNameSize

This int value specifies the size of the buffer storing the name of the remote model.

DESCRIPTION

DS_getInEntityName is used to get the entity name and the source model name for a specified entityID.

RETURN VALUES

The return address points to a buffer storing the name of the entity with the specified entityID. NULL is returned if an entity with the specified entityID does not exist.

HLA FUNCTIONS INVOKED

None.

A.4.3 DS_getOutEntityID()

ABSTRACT

DS_getOutEntityID tries to get the entityID with the specified entity name and the model name to which the entity is sent. This entity must have been registered as one that will be sent by the model.

SYNOPSIS

```
Procedure DS_getOutEntityID (
        const char* entityName,
        int entityNameSize,
        const char* destinationModelName,
        int destinationModelNameSize)
        returning int;
```

ARGUMENTS

entityName

This address points to a buffer storing the name of the entity which will be sent to other remote models in the distributed simulation.

EntityNameSize

This int value specifies the size of the buffer storing the name of the entity.

DestinationModelName

This address points to a buffer storing the name of the remote model to which the registered entity will be sent.

DestinationModelNameSize

This int value specifies the size of the buffer storing the name of the remote model.

DESCRIPTION

DS_getOutEntityID is used to get the unique entityID for the specified entity name and the model name to which the entity is sent. The entityID for OutEntity is used transfer the entity when invoking DS_setAttributeValue and DS_transferEntity.

RETURN VALUES

The return value is –1 if an entity with the specified entity name and destination model name does not exist. Or else the positive value is a unique entityID.

HLA FUNCTIONS INVOKED

None.

A.4.4 DS_getOutEntityName()

ABSTRACT

DS_getOutEntityName tries to get the entity name and the model name to which the entity is sent with the specified entityID. This entity must have been registered as one that will be sent to other models.

SYNOPSIS

```
Procedure DS_getOutEntityName (
        int entityID,
        int entityNameSize,
        char** destinationModelName
        int destinationModelNameSize)
returning char*;
```

ARGUMENTS

entityID

This int value specifies the entityID.

EntityNameSize

This int value specifies the size of the buffer containing the name of the entity.

DestinationModelName

This address points to a buffer storing the name of the remote model to which the registered entity will be sent.

DestinationModelNameSize

This int value specifies the size of the buffer storing the name of the remote model.

DESCRIPTION

DS_getOutEntityName is used to get the entity name and the destination model name for a specified entityID.

RETURN VALUES

The return address points to a buffer storing the name of the entity with the specified entityID. NULL is returned if an entity with the specified entityID does not exist.

HLA FUNCTIONS INVOKED

None.

APPENDIX B: DSMANAGER STRUCTURE

As part of the middleware between the CSP and the HLA, the DSManager has all the necessary interoperability information provided by the CSP as well as the callback information returned from the underlying HLA RTI+. The DSManager stores and processes this information, and provides a generic interface to the CSP.

B.1 The SysManager Class

The HLA uses an object-oriented (OO) view on simulations and simulated entities. Moreover, the *RTIAmbassador* and *FederateAmbassador* are also in the form of classes with member functions used by the simulation model. Some CSPs, however, cannot support OOP (Object Oriented Programming) or do not allow a callback mechanism. To provide a generic approach for the integration of the CSP with the HLA, the interface of the DSManager consists of a set of one-directional pure C functions. This interface is defined by invoking the public functions of a class named *SysManager* (see Figure B.1). Moreover, the *SysManager* also contains some variables and other supporting functions to handle the details of communication with the HLA.

The purposes of these variables are explained as follows:

- The two pointers *rtiAmb* and *fedAmb* are used to communicate with the RTI.
- The variables, *joinedDS*, *isController*, *nFederates*, *localModelName*, *federationName*, *synMethod*, *simEndTime* and *lookahead*, are provided by the CSP based on the model properties.
- The variables, *nOutEntity* and *nRestriInformation* are used to check whether all destination models (to which entities will be transferred from the local model) have sent *RestriInfo* to the local model. When *nRestriInformation* is equal to *nOutEntity*, another variable, *startDS*, is set as true since the local model has received *RestriInfo* from all destination models. Note that *startDS* is set to true as default for Type I IRM. Based on the *RestriInfo*, the SysManager will decide whether near zero lookahead must be set and whether the status should be updated dynamically (more details are given in section 3.4).
- The two lists, *DSSubObjectList* and *DSPubObjectList*, are defined to store all the entity information exchanged with other models in the distributed simulation. There is some information associated with the entities, e.g., status and priority information for Type II IRM, which must be updated during the simulation. Such information is stored and maintained by two other lists, *sysDSSubObjectList* and *sysDSPubObjectList*.
- The variables, *DSTimeRegualtion*, *DSTimeConstrained*, *objectFound*, *DSTimeAdvGrant*, and *grantTime* are used to store the callback information returned from *federateAmbassador*. *nDSEntityReceived* is the number of entities received during the current time advancement request.
- The variables, *needZeroLookahead*, *needSlightIncrease*, *hiddenField* and *newUpdatedStatus* are specially defined for the Type II IRM and may be applied to other types of IRMs in the future. Currently they are used to solve the problems introduced by synchronous entity

transfer. Since the hidden field is transparent to the CSP, another variable *modelTime*, without the hidden field is used to represent the time returned to the CSP.

SysManager
RTIAmbassadorPlus *ritAmb FederateAmbassdorPlus *fedAmb bool joinedDS bool isController int nFederates char*localModelName char*federationName int synMethod double simEndTime double lookahead int nOutEntity int nRestriInformation bool startDS vector <DS_SysObject> sysDSSubObjectList vector <DS_SysObject> sysDSPubObjectList vector <DS_Object> DSSubObjectList vector <DS_Object> DSPubObjectList RTIfedTime grantTime RTI::Boolean DSTimeAdvGrant RTI::Boolean DSTimeRegulation RTI::Boolean DSTimeConstrained RTI::Boolean objectFound int nDSEntityReceived double modelTime bool needZeroLookahead bool needSlightIncrease int hiddenField bool newUpdatedStatus
bool isControllerModel() int getNoOfModels() double getLookahead() int getSynMethod() double getSimEndTime() void init(RTI::RTIambassadorPlus *rtiAmb, ModelFederateAmbassadorPlus **fedAmb, const char** DSName, const char* ModelName) void add_federate() int get_nFederate() RTI::ObjectClassHandle get_sysObjHDL() void clear() void SysReceiveEntity(RTI::InteractionClassHandle theInteraction, const RTI::ParameterHandleValuePairSet& p, const RTI::FedTime& theTime, const char *theTag, RTI::EventRetractionHandle hdl) void SysReceiveEntity(RTI::InteractionClassHandle theInteraction, const RTI::ParameterHandleValuePairSet& p, const char *theTag) double getModelTime() void updateExEntryStatus() + int registerController(int noOfModels) + int registerDistributedSimulation(const char* DSName, const char* modelName, const char* fedFileName) + int registerDistributedSimulationWithTP(const char* DSName, const char* modelName, const char* fedFileName, int timeConstrained, int timeRegulating) + int registerInEntity(const char* entityName, const char* sourceModelName) + int registerOutEntity(const char* entityName, const char* destinationModelName) + void setSynMethod(int method, double endTime) + void setLookahead(double la) + double getLookahead() + double advanceTime(double theTime, int * nEntityRecived) + void setAttributeValue(int entityID, const char* AttriName, const char* value, int valueSize) + char* getAttributeValue(int entityID, const char* AttriName, int* valueSize) + int transferEntity(int entityID, double time) + int transferEntity(int entityID, double time, double travelTime) + int receiveEntity(int entityID, double* time) + int getInEntityID(const char* entityName, const char* sourceModelName) + int getOutEntityID(const char* entityName, const char* destinationModelName) + char* getInEntityName(int entityID, char** sourceModelName) + char* getOutEntityName(int entityID, char** destinationModelName) + int getExEntryStatus(const char* entityName, const char* exModelName) + int setExEntryStatus(const char* entityName, const char* exModelName, int status) + void terminateDS()

Figure B.1: SysManager class

The SysManager also has some supporting functions to process the variables and facilitate the operation of the public functions. Among them, *SysReceiveEntity()* with the parameter of *theTime* is used to receive an interaction sent from other models, including the entity information as well as the status and priority information. The other one, *SysReceiveEntity()* without the parameter of *theTime* is for receiving *RestriInfo* in the initialization phase. The function of *updateExEntryStatus()* is also defined for the Type II IRM, which is invoked after each time advancement request since possible new status or priority information may be received in this procedure.

The functions beginning with the symbol "+" are public, consistent with the interface of the DSManager shown to the CSP. Currently they can support Type I and Type II IRMs and will be updated for other types of IRMS in future research work.

B.1.1 The DS_Entity Class and the SysDS_Entity Class

As discussed, the interface is based on the entity transfer mechanism. Each entity is transferred using a timestamped interaction. The DS_*Entity* class (see Figure B.2) is used to describe each type of entity (represented by a different entity name). It is possible that an entity of the same type is received from or sent to multiple external models. The information of the external models is kept in *exModelList*. Similar to the *DS_Entity* class, the *SysDS_Entity* class (see Figure B.3) is used to represent the additional information for each entity type, such as priority and status information for Type II IRMs. It also has a list *exModelList* to contain the information of external models.

DS_Entity
string entityName vector <DS_ExModel> exModelList

Figure B.2: DS_Entity class

SysDS_Entity
String entityName vector <SysDS_ExModel> exModelList

Figure B.3: SysDS_Entity class

B.1.2 The DS_ExModel Class and the SysDS_ExModel Class

The *DS_ExModel* class (see Figure B.4) is defined for each external model to which entities will be sent or received. It has some variables to contain the information of the external model as shown below:

- *modelName* is the name of the external model.

- *classHandle* keeps the returned value from the RTI when *subscribeInteractionClass* or *publishInteractionClass* is invoked for future entity transfer. It will be used to send or receive interactions during the simulation run.
- *newEntities* counts the number of entities of the specific type from this external model. When each entity with the specific type is received from this external model by the *SysManager*, the value of *newEntities* will be increased by 1. Contrarily, the value of *newEntities* will be decreased by 1 when such an entity is passed to the CSP.
- *attrisList* is used to store the information of the attributes.
- *time* is the time when this entity will be received from or sent to this external model.
- *travelTimeList* stores the travel time (simulation time) needed to transfer the entity. The travel time is defined as the difference between the time when the entity is received in the destination model and the time when the entity is sent from the source model.

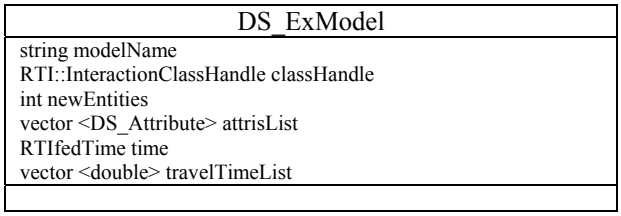


Figure B.4: DS_ExModel class

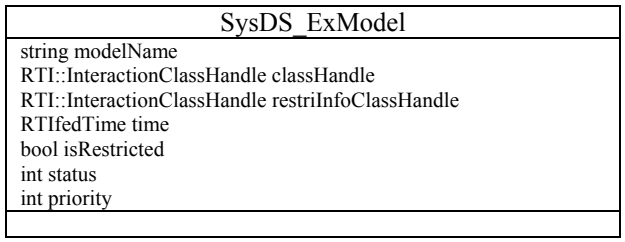


Figure B.5: SysDS_ExModel class

Similar to the DS_EXModel class, another class named SysDS_ExModel (see Figure B.5) is used to record extra information from the external model. The meaning of its variables are explained as follows:

- *modelName* is the name of the external model.
- *classHandle* keeps the returned class handle from the RTI when *subscribeInteractionClass* or *publishInteractionClass* is invoked for status and priority information.
- *restriInfoClassHandle* keeps the returned class handle from the RTI when *subscribeInteractionClass* or *publishInteractionClass* is invoked for *RestriInfo*.
- *time* is the time when this entity will be received from or sent to this external model.

- *isRestricted* stores the *RestriInfo* received from this external model for the specific entity type. It indicates whether the entity will be sent to a bounded queue or a workstation with limited capacity in this external model.
- *status* and *priority* contains the status and priority information for some remote EEP in this external model.

B.1.3 The DS_Attribute Class and the DS_AttriValue Class

Each entity may have zero or more attributes. The *DS_Attribute* class (see Figure B.6) is used to hold the information of each attribute. The variable *name* is the name of the attribute. The reason why a list *attriValueList* is defined to keep the attribute values is because it is possible more than one entity of a specific entity type from a specific external model is received at the same simulation time during the time advancement request. The *SysManager* will forward different values for different entities. Note that if one entity of a specific type has more than one attribute and a set of such entities is received, the attribute values for different attributes should be kept in the same order.

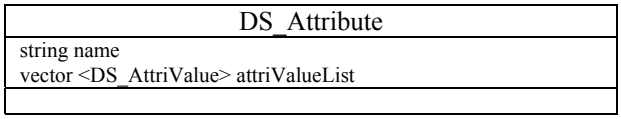


Figure B.6: DS_Attribute class

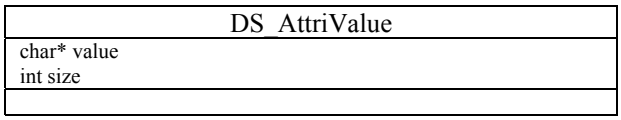


Figure b.7: DS_AttriValue class

In the HLA, the information is sent as a stream of bytes between the federates. The encoding and decoding process should be performed by the federates themselves. It is difficult for the DSManager to perform this task on behalf of the CSP since it has no idea about the data type for each attribute. Moreover, the representation of each data type may differ in different systems. So the *DS_AttriValue* class (see Figure B.7) is defined with two variables: *value* is a buffer to store the attribute value and *size* is the length of this buffer. The CSP should be able to interpret the received value for its own purpose.

APPENDIX C: AUTHOR'S PUBLICATION

Journals:

[1] **Wang, X.G.**, S.J. Turner, M.Y.H. Low, B.P. Gan, "Optimistic Synchronization in HLA Based Distributed Simulation". *Simulation: Transactions of the Society for Modeling and Simulation International*, Special Issue "Best of PADS 2004", Vol. 81, No. 4, Apr. 2005, pp. 279-293.

[2] Taylor, S.J.E., **X.G. Wang**, S.J. Turner, M.Y.H. Low, "Integrating Heterogeneous Distributed COTS Discrete-Event Simulation Packages: An Emerging Standards-Based Approach". *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 36, No. 1, Jan. 2006, pp. 109-122.

[3] Low, M.Y.H., B.P. Gan, J.H. Wei, **X.G. Wang**, S.J. Turner, W.T. Cai, "Shared State Synchronization for HLA-Based Distributed Simulation". Accepted by *Simulation: Transactions of the Society for Modeling and Simulation International*, 2006.

Conferences:

[1] **Wang, X.G.**, S.J. Turner, M.Y.H. Low and B.P. Gan, "A Generic Architecture for the Integration of COTS Packages with the HLA". *UK Operational Research Society Simulation Workshop, Birmingham*, UK, Mar. 23-24, 2004, pp. 225-233.

[2] **Wang, X.G.**, S.J. Turner, M.Y.H. Low and B.P. Gan, "Optimistic Synchronization in HLA Based Distributed Simulation". In *Proceedings of the 18th IEEE/ACM/SCS Workshop on Parallel and Distributed Simulation (PADS 2004)*, IEEE Computer Society, Kufstein, Austria, May 16-19, 2004, pp. 225-233.

[3] **Wang, X.G.**, S.J. Turner, M.Y.H. Low and B.P. Gan, "A COTS Simulation Package Emulator for Investigating COTS Simulation Package Interoperability". In *Proceedings of the 2005 Winter Simulation Conference*, Orland Florida, USA, Dec. 4-7, 2005, pp. 402-411.

[4] **Wang, X.G.**, S.J. Turner and S.J.E. Taylor, "COTS Simulation Package (CSP) Interoperability – A Solution to Synchronous Entity Passing". In *Proceedings of the 20th IEEE/ACM/SCS Workshop on Principles of Advanced and Distributed Simulation (PADS 2006)*, IEEE Computer Society, Singapore, May 23-26, 2006, pp. 201-210.

[5] Gan, B.P., M.Y.H. Low, J.H. Wei, **X.G. Wang**, S.J. Turner and W.T. Cai, "Synchronization and Management of Shared State in HLA-Based Distributed Simulation". In *Proceedings of the 2004 Winter Simulation Conference*, New Orleans, USA, Dec. 7-10, 2003, pp. 847-854.

[6] Low, M.Y.H., B.P. Gan, J.H. Wei, **X.G. Wang**, S.J. Turner and W.T. Cai, "Implementation Issues for Shared State in HLA-Based Distributed Simulation". In *Proceedings of the 2003 European Simulation Symposium*, Delft, The Netherlands, Oct. 26-29, 2003, pp. 5-13.

[7] Gan, B.P., P. Lendermann, M.Y.H. Low, S.J. Turner, **X.G. Wang** and S.J.E. Taylor, "Interoperating AutoSched AP Using the High Level Architecture". In *Proceedings of the 2005 Winter Simulation Conference*, Orland Florida, Dec. 4-7, USA, 2005, pp. 394-401.

[8] Lendermann, P, M.Y.H. Low, B.P. Gan, N. Julka, L.P. Chan, S.J. Turner, W.T. Cai, **X.G. Wang**, L.H. Lee, T. Hung, S.J.E. Taylor and L.F. McGinnis, "An Integrated and Adaptive Decision-Support Framework for High-Tech Manufacturing and Service Networks". In *Proceedings of the 2005 Winter Simulation Conference*, Orland Florida, Dec. 4-7, USA, 2005, pp. 2052-2062.

[9] Gan, B.P., M.Y.H. Low, S.J. Turner and **X.G. Wang**, "Using Manufacturing Process Flow for Time Synchronization in HLA-Based Simulation". In *Proceedings of the 9th IEEE International Symposium on Distributed Simulation and Real Time Applications*, Montreal, Qc. Canada, Oct. 10-12, 2005, pp. 148-157.

[10] Taylor, S.J.E., L. Bøhli, **X.G. Wang**, S.J. Turner and J. Ladbrook, "Investigating Distributed Simulation at the Ford Motor Company". In *Proceedings of the 9th IEEE International Symposium on Distributed Simulation and Real Time Applications*, Montreal, Qc. Canada, Oct. 10-12, 2005, pp. 139-147.

[11] Gan, B.P., S.J.E. Taylor, S.J. Turner, **X.G. Wang** and J. Zhang, "Developing Standards to Support Distributed Simulation in Semiconductor Manufacturing". In *Proceedings of International Conference on Modeling and Analysis of Semiconductor Manufacturing (MASA2005)*, Singapore, 2005, pp. 373-380.

BIBLIOGRAPHY

[ALOGIC] XJ Technologies Company. Anylogic. Available via <http://www.xjtek.com/products/anylogic>.

[ARENA] Rockwell Automation, Inc., Arena Simulation. Available via <http://www.arenasimulation.com>.

[ASAP] Brooks Automation. Inc., AutoSched AP. Available via <http://www.brooks.com/pages/231-autosched_ap.cfm>.

[BAL06] Peter Ball, "Introduction to Discrete Event Simulation". In *2nd DYCOMANS workshop on "Management and Control: Tools in Action"*, the Algarve, Portugal, May 15-17, 1996, pp. 367-376.

[BAS88] Bain, William L. and David S. Scott, "An Algorithm for Time Synchronization in Distributed Discrete Event Simulation". In *Proceedings of the SCS Multiconference on Distributed Simulation*, February 3-5, 1988, 19(3): 30-59.

[BAS03] Bapat, V. and D.T. Sturrock, "The Arena Product Family: Enterprise Modeling Solutions". In *Proceedings of the 2003 Winter Simulation Conference*, New Orleans, LA, USA, Dec. 7-10, 2003, pp. 210-217.

[BEL90] Bellenot, S., "Global Virtual Time Algorithms". In *Proceedings of the 1990 SCS Multiconference on Distributed Simulation*, Society for Computer Simulation, San Diego, USA, Jan. 1990, pp. 122-127.

[BEL92] Bellenot, S., "State Skipping Performance with the Time Warp Operating System". In *Proceedings of the 1992 SCS Workshop on Parallel and Distributed Simulation*, Society for Computer Simulation, 1992, pp. 53-61.

[BKK02] Borshchev, Andrei, Yuri Karpov, Vladimir Kharitonov, "Distributed Simulation of Hybrid Systems with AnyLogic and HLA". Future Generation Computer System, 2002, 18(6): 829-839.

[BLE] Handling big/little endian issues.
Available via <http://www.home.ix.netcom.com/~stonybrk/env/index648.htm>.

[BRY77] Bryant, R.E., "Simulation of Packet Communication Architecture Computer Systems". Technical Report MIT-LCS-TR-188, Massachusetts Institute of Technology, 1977.

[BSS98] Bluemel, E., M. Schumann, T. Schulze, "Using HLA and SLX for Logistical Simulation". In *Proceedings of the International Workshop on Modeling and Simulation Within a Maritime Environment*, Riga, Latvia, Sep. 6-8, 1998, pp. 76-85.

[CAT90] Cai, Wentong and Stephen J. Turner, "An Algorithm for Distributed Discrete-event Simulation – The 'Carrier Null Message' Approach". In *Proceedings of the SCS Multiconference on Distributed Simulation*, January 1990, 22(1): 3-8.

[CDM79] Chandy, K. Mani and Jayadev Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs". *IEEE Transactions on Software Engineering*, Sep. 1979, SE-5 (5): 440-452.

[CHT98] Choe, M. and C. Tropper, "An Efficient GVT Computation Using Snapshots". *In Proceedings of the Conference on Computer Simulation Methods and Applications*, Society for the Computer Simulation, Nov. 1998, pp. 33-43.

[CLT97] Cai, Wentong, E. Letertre and S. J. Turner, "Dag Consistent Parallel Simulation: a Predictable and Robust Conservative Algorithm". In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation (PADS'97)*, June 1997, pp.178-181.

[COR90] Cota, Bruce A., Robert G. Sargent, "Simultaneous Events and Distributed Simulation". In *Proceedings of the 1990 Winter Simulation Conference*, New Orleans, LA, USA, Dec. 9-12, 1990, pp. 436-440.

[COS90] Cota, Bruce A. and Robert G. Sargent, "A Framework for Automatic Lookahead Computation in Conservative Distributed Simulations". In *Proceedings of the SCS Multiconference on Distributed Simulation*, January 1990, 22(1): 56-59.

[COX98] Cox, Kevin, "A Framework-based Approach to HLA Federate Development". In *Fall Simulation Interoperability Workshop*, Orlando, Sep. 14-18, 1998, pp. 1059-1068.

[DBS01] Deelman, Ewa, Rajive Bagrodia, Rizos Sakellariou and Vikram Adve, "Improving Lookahead in Parallel Discrete Event Simulations of Large-Scale Applications using Compiler Analysis". In *Proceedings of the 15th Workshop on Parallel and*

Distributed Simulation, Lake Arrowhead, California, May 15-18, 2001, pp. 5-13.

[DIR90] Dickens, P.M. and P.F. Reynolds, "SRADS with Local Rollback". In *Proceedings of the SCS Multiconference on Distributed Simulation*, 1990, 22, 1, pp. 161- 164.

[DIS94] "Standard for Distributed Interaction Simulation – Application Protocols (Version 2.0 Fourth Draft)". University of Central Florida Institute for Simulation and Training, IST-CR-94-15, February 1994.

[DMSO00] The Defense Modeling and Simulation Office (DMSO) and Object Management Group, Inc. "Distributed Simulation Systems Specification". Version 1.0, April 2000.

[DOD98] High Level Architecture Federation Execution Data (FED) File Specification – RTI 1.3 Version 3. Department of Defense. 31 July, 1998.

[DSS00] The Defense Modeling and Simulation Office (DMSO) and Object Management Group, Inc. "Distributed Simulation Systems Specification". Version 1.0, April 2000.

[DWG97] Damani, O. P., Yi-Min Wang, V. K. Garg, "Optimistic Distributed Simulation Based on Transitive Dependency Tracking". In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation (PADS'97)*, Lockenhaus, Austria, June 1997, pp. 90-97.

[FDK] Georgia Tech University. "FDK – Federated Simulations Development Kit". Available via <http://www.cc.gatech.edu/computing/pads/fdk>.

[FDP97] Fujimoto, R. M., Samir Das, Kiran Panesar, Maria Hybinette and Chris Carothers, "Georgia Tech Time Warp (GTW Version 3.1) Programmer's Manual for Distributed Network of Workstations". Technical Report, July 1997.

[FER95] Ferscha, Alois, "Parallel and Distributed Simulation of Discrete Event Systems". *In Handbook of Parallel and Distributed Computing*. McGraw-Hill, 1995.

[FFR94] Feigin, G., J. Fowler, J. Robinson and R. Leachman, "*Semiconductor Wafer Manufacturing Data Format Specification*". SEMATECH, 1994.

[FJT01] Ferscha, Alois, James Johnson and Stephen J. Turner, "Distributed simulation performance data mining". *Future Generation Computer Systems*, 18 (2001), pp: 157–174.

[FLEX] Flexsim Software Products, Inc. Available via <http://www.flexsim.com/>.

[FLP02] Fujimoto, R., D. Lunceford, E. Page and A. Uhrmacher, "Technical Report of the Dagstuhl-Seminar Grand Challenges for Modelling and Simulation". Available via <http://www.dagstuhl.de/02351/Report/.2002>.

[FPF00] Ferenci, S.L., K.S. Perumalla, R.M. Fujimoto, "An Approach for Federating Parallel Simulators". In *Proceedings of the 14th Workshop on Parallel and Distributed Simulation (PADS 2000)*, May 28-31, 2000, Bologna, Italy, pp. 63-70.

[FUJ89] Fujimoto, R. M., "Time Warp on a Shared Memory Multiprocessor". *Transactions of the Society for Computer Simulation*, July 1989, 6(3): 211-239.

[FUJ90] Fujimoto, R. M., "Parallel Discrete Event Simulation". *Communications of ACM*, 33, no.10 (October), 1990, pp. 30-53.

[FUJ98] Fujimoto, R. M., "Time Management in the High Level Architecture". *SCS Simulation Magazine,* December 1998.

[FUJ00] Fujimoto, R. M., "Parallel and Distributed Simulation Systems". *Wiley Interscience*, January 2000.

[FUJ01] Fujimoto, R. M., "Parallel and Distributed Simulation Systems". In *Proceedings of the 2001 Winter Simulation Conference*, Arlington, VA, USA, Dec. 9-12, 2001, pp. 147-157.

[FUW96] Fujimoto, R. M. and R. M. Weatherly, "HLA Time Management and DIS". In *Proceedings of the 14th Workshop on Standards for Interoperability of Distributed Simulation*, March 1996.

[GAF88] Gafni, Anat, "Rollback Mechanisms for Optimistic Distributed Simulation Systems". In *Proceedings of the SCS Multiconference on Distributed Simulation*, February 1988, 19(3): 61-67.

[GKP96] Geist, G., J. Kohl, P. Papadopoulos, "PVM and MPI: A Comparison of Features", 1996. Available via <http://www.epm.ornl.gov/pvm/PVMvsMPI.ps>.

[GLT05] Gan, B.P., M.Y.H. Low, S.J. Turner, X.G. Wang, and S.J.E. Taylor, " Interoperating Autosched AP using the High Level Architecture". In *Proceedings of the 2005 Winter Simulation Conference*, Orlando, USA, Dec. 4-7, 2005.

[GLJ00] Gan, B.P., L. Liu, S. Jain, S.J. Turner, W. Cai and W.J. Hsu, "Distributed Supply Chain Simulation Across the Enterprise Boundaries". In *Proceedings of the 2000 Winter Simulation Conference*, Orlando, FL, USA, Dec. 10-13, 2000, pp: 1245-1251.

[GLW03] Gan, B.P., M.Y.H. Low, J.H. Wei, X.G. Wang, S.J. Turner and W.T. Cai, "Synchronization and Management of Shared State in HLA-Based Distributed Simulation". In *Proceedings of the 2003 Winter Simulation Conference*, New Orleans, Louisiana, USA, Dec. 7-10, 2003, pp. 847-854.

[GTT05] Gan, B.P., S.J.E. Taylor, S.J. Turner, X.G. Wang and J. Zhang, "Developing Standards to Support Distributed Simulation in Semiconductor Manufacturing". In *Proceedings of International Conference on Modeling and Analysis of Semiconductor Manufacturing (MASM2005)*, Singapore, 2005, pp. 373-380.

[HAF01] Harrell, Charles R. and Kevin C. Field, "ProModel / MedModel: Simulation Modeling and Optimization Using ProModel Technology". In *Proceedings of the 2001 Winter Simulation Conference*, Arlington, VA, USA, Dec. 9-12, 2001, pp. 226-232.

[HAP00] Harrell, Charles R. and Rochelle N. Price, "ProModel / MedModel: Simulation Modeling and Optimization Using ProModel". In *Proceedings of the 2000 Winter Simulation Conference*, Orlando, FL, USA, Dec. 10-13, 2000, pp. 197-202.

[HAP03] Harrell, C.R. and R.N. Price, "Simulation Modeling Using ProModel Technology". In *Proceedings of the 2003 Winter Simulation Conference*, New Orleans, LA, USA, Dec. 7-10, 2003, pp. 175-181.

[HEC95] Henriksen, J.O. and R.C. Crain, "GPSS/H Professional System Guide", *Wolverine Software Corporation*, 1995.

[HEN95] Henriksen, James O., "An Introduction to SLX". In *Proceedings of the 1995 Winter Simulation Conference*, Arlington, VA, USA, Dec. 3-6, 1995, pp. 502-509.

[HEN96] Henriksen, James O., "An Introduction to SLX". In *Proceedings of the 1996 Winter Simulation Conference*, Coronado, CA, USA, Dec. 8-11, 1996, pp. 468-475.

[HEN97] Henriksen, James O., "An Introduction to SLX". In *Proceedings of the 1997 Winter Simulation Conference*, Atlanta, GA, USA, Dec. 7-10, 1997, pp. 559-566.

[HEN99] Henriksen, James O., "SLX: Pyramid Power". In *Proceedings of the 1999 Winter Simulation Conference*, Phoenix, AZ, USA, Dec. 5-8, 1999, pp. 167-175.

[HEN00] Henriksen, James O., "SLX: The X is For Extensibility". In *Proc. 2000 Winter Simulation Conference*, Orlando, FL, USA, Dec. 10-13, 2000, pp. 183-190.

[HIF02] Hibino, Hironori and Yoshiro Fukuda, "Manufacturing Adapter of Distributed Simulation Systems Using HLA". In *Proceedings of the 2002 Winter Simulation Conference*, San Diego, CA, USA, Dec. 8-11, 2002, pp. 1099-1107.

[HLA-CSPIF] High Level Architecture Commercial Off-The-Shelf Simulation Package Interoperation Forum (HLA-CSPIF). Available via <http://www.cspif.com>.

[HUL03] Huang, Paul, Young M. Lee, Lianjun An, Markus Ettl, Steve Buckley and Karthik Sourirajan, "Utilizing Simulation to Evaluate Business Decisions in Sense-and-Respond Systems". In *Proceedings of the 2004 Winter Simulation Conference*, New Orleans, USA, Dec. 7-10, 2003, pp. 1205-1212.

[IEEE1516] IEEE 1516, "Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Framework and Rules". 2000.

[IEEE1516.1] IEEE 1516.1, "Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Federate Interface Specification". 2000.

[IEEE1516.2] IEEE 1516.2, "Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) –HLA Object Model Template (OMT)". 2000.

[IEEE1516.3] IEEE 1516.3, "High Level Architecture (HLA) Federation Development and Execution Process (FEDEP)". April 2003.

[JEF85] Jefferson, D., "Virtual Time". *ACM Transactions on Programming Languages and Systems*, 1985, 7(3): 404-425.

[JES82] Jefferson, D. and H. Sowizral, "Fast Concurrent Simulation Using the Time Warp Mechanism; Part I: Local Control". Tech. Rep. N1906AF.RAND Corporation, December 1982.

[JOH99] Johnson, Glen D., "Networked Simulation with HLA and MODSIM III". In *Proceedings of the 1999 Winter Simulation conference*, Phoenix, AZ, USA, Dec. 5-8, 1999, pp. 1065-1070.

[KRA03] Krahl, D., "Extend: An Interactive Simulation Tool". In *Proceedings of the 2003 Winter Simulation Conference*, New Orleans, LA, USA, Dec. 7-10, 2003, pp.188-196.

[KUS05] Kusuma, H.A., "Integrating a Commercial Off-the-Shelf Simulation Package with a Business Application". Final Year Project Report, Nanyang Technological University, 2005.

[KWD99] Kuhl, F., R. Weatherly and J. Dahmann, "Creating Computer Simulation Systems: An Introduction to the High Level Architecture". *Prentice Hall,* 1999.

[LAC06] Lacy, Lee W., "Interchanging Discrete Event Simulation Models Using PIMODES and SRML". In *Proceedings of 2006 Fall Simulation Interoperability Workshop*, 06F-SIW-047, Orlando, Florida, USA, Sep. 10-15, 2006.

[LEN06] Lendermann, P., "About the Need for Distributed Simulation Technology for the Resolution of Real-world Manufacturing and Logistics Problems". To appear in *Proceedings of the 2006 Winter Simulation Conference*, Monterey, CA, USA, Dec. 3-6, 2006.

[LGL04] Lendermann, P., B.P. Gan, Y.L. Loh, H.K. Tan, S.K. Lieu, L.F. McGinnis and J.W. Fowler, "Analysis of a Borderless Fab Scenario in a Distributed Testbed". In *Proceedings of the 2004 Winter Simulation Conference*, New Orleans, USA, Dec. 7-10, 2003, pp. 1896-1901.

[LGW03] Low, M.Y.H., B.P. Gan, J.H. Wei, X.G. Wang, S.J. Turner and W.T. Cai, "Implementation Issues for Shared State in HLA-Based Distributed Simulation". In *Proceedings of the 2003 European Simulation Symposium*, Delft, The Netherlands, October 26-29, 2003, pp. 5-13.

[LGW06] Low, M.Y.H., B.P. Gan, J.H. Wei, X.G. Wang, S.J. Turner and W.T. Cai, "Shared State Synchronization for HLA-Based Distributed Simulation". Accepted by *Simulation: Transactions of the Society for Modeling and Simulation International*, 2006.

[LIT90] Liu, L.Z. and C. Tropper, "Local Deadlock Detection in Distributed Simulations", In *Proceedings of the SCS Multiconference on Distributed Simulation*, SCS Simulation Series 22, 1990, pp. 64-69.

[LJC03] Lendermann, P., N. Julka, L.P. Chan and B.P. Gan, "Integration of Discrete Event Simulation Models with Framework-based Business Applications". In *Proceedings of the 2003 Winter Simulation Conference*, New Orleans, Louisiana, USA, Dec. 7-10, 2003, pp. 1797-1804.

[LJG03] Lendermann. P., N. Julka, B.P. Gan, D. Chen, L.F. McGinnis and J.P. McGinnis, "Distributed Supply Chain Simulation as a Decision Support Tool for the Semiconductor Industry". *Simulation* 79:1, pp. 126-138, 2003.

[LLG98a] Low, Yoke-Hean, Chu-Cheow Lim, Boon-Ping Gan, Sanjay Jain, Wentong Cai, Wen Jing Hsu, Shell Ying Huang and Stephen J. Turner, "Conservative Parallel Simulation for Manufacturing Systems". In *Proceedings of the 8th International Parallel Computing Workshop (PCW'98)*, Singapore, September 1998, pp. 293–300.

[LLG98b] Lim, Chu-Cheow, Yoke-Hean Low, Boon-Ping Gan and S. Jain, "Implementation of Dispatch Rules in Parallel Manufacturing Simulation". In *Proceedings of the 1998 Winter Simulation Conference*, Washington DC, USA, Dec. 13-16, 1998, pp. 1591-1597.

[LLG99] Lim, Chu-Cheow, Yoke-Hean Low and Boon-Ping Gan, "Computing Safetime in a Conservative Synchronous Simulation Based on Future Events". In *1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA99)*, Las Vegas, Nevada, USA, Jun. 28 – Jul. 1, 1999, pp. 2436-2442.

[LLG05] Lendermann, P, M.Y.H. Low, B.P. Gan, N. Julka, L.P. Chan, S.J. Turner, W.T. Cai, X.G. Wang, L.H. Lee, T. Hung, S.J.E. Taylor and L.F. McGinnis, "An Integrated and Adaptive Decision-Support Framework for High-Tech Manufacturing and Service Networks". In *Proceedings of the 2005 Winter Simulation Conference*, Orland Florida, Dec. 4-7, USA, 2005.

[LLT98] Lim, Chu-Cheow, Yoke-Hean Low and Stephen J. Turner, "Relaxing Safetime Computation of a Conservative Simulation Algorithm". In *1998 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA98)*, Las Vegas, Nevada, USA, Jul. 13–16, 1998, pp. 1538-1545.

[LWS89] Lubachevsky, B., A. Weiss and A. Shwartz, "Rollback Sometimes Works . . . If Filtered". In *Proceedings of the 1989 Winter Simulation Conference*, Washington DC, USA, Dec. 4-6, 1989, pp. 630-639.

[LUB88] Lubachevsky, Boris D., "Efficient Distributed Event-Driven Simulation". In *Proceedings of the SCS Multiconference on Distributed Simulation*, February 1988, 19(3): 183-190.

[MAD88] Madisetti, Vijay, "WOLF: A Rollback Algorithm for Optimistic Distributed Simulation Systems". In *Proceedings of the 1988 Winter Simulation Conference*, San Diego, California, Dec. 12-14, 1988, pp. 296-305.

[MAM97] Markt, P.L. and M. H. Mayer, "Witness Simulation Software A Flexible Suite of Simulation Tools". *In Proc. 1997 Winter Simulation Conference*, Georgia, USA, Dec. 7-10, 1997, pp. 711-717.

[MAT93] Mattern, F., "Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation". *Journal of Parallel and Distributed Computing*, 1993, 18(4): 423-434.

[MEB99] Meyer, Richard A., Rajive L. Bargrodia, "Path Lookahead: a Data Flow View of PDES Models". In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, Atlanta, Georgia, United States, May 1-4, 1999, pp. 12-19.

[MEH92] Mehl, H., "A Deterministic Tie-Breaking Scheme for Sequential and Distributed Simulation". In *Proceedings of the 6th Workshop on Parallel and Distributed Simulation*, CA, USA, Jan. 20-22, 1992, pp. 199-200.

[MEH93] Mehl, H. and S. Hammes, "Shared Variables in Distributed Simulation". In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, San Diego, California, May 16-19, 1993, pp. 68-75.

[MFT01] Miller, J.A., P.A. Fishwick, S.J.E. Taylor, B. Benjamin and B. Szymanski, "Research and Commercial Opportunities in Web-Based Simulation". *Simulation: Practice and Theory*, 9(1-2), 2001.

[MIS86] Misra, Jayadev, "Distributed Discrete-Event Simulation". *ACM Computing Surveys*, March 1986, 18(1): 39-65.

[MOT02] M. Morisio, and M. Torchiano, "Definition and Classification of COTS: A Proposal". In *Proceedings of the 1st International Conference on COTS-Based Software Systems* (ICCBSS 2002), LNCS 2255, Springer-Verlag, 2002, pp. 165-175.

[MPI] Message Passing Interface. Available via <http://www.mpi-forum.org>.

[NIC88] Nicol, David M., "Parallel Discrete-Event Simulation of FCFC Stochastic Queuing Networks". *ACM SIGPLAN Notices Proceedings of PPEALS, New Haven, Connecticut*, 1988, pp. 23(9): 124-137.

[NIC93] Nicol, David M., "The Cost of Conservative Synchronization in Parallel Discrete-event Simulation". *J. ACM*, vol. 40, 2, Apr., 1993, pp. 304-333.

[NIR84] Nicol, David M. and Paul F. Reynolds, Jr., "Problem Oriented Protocol Design". In *Proceedings of the 1984 Winter Simulation Conference*, Sheraton Dallas Hotel, Dallas, TX, USA, Nov. 28-30, 1984, pp. 471-474.

Dallas, TX, USA, Nov. 28-30, 1984, pp. 471-474.

[PID04] Pidd, M., "Computer Simulation in Management Science", *Wiley*, 5th edition, 2004.

[PLD00] Pawletta, S., B. Lampe, W. Drewelow and T. Pawletta, "HLA-based Simulation within an Interactive Engineering Environment". In *Proceedings of the 4th International Workshop on Distributed Simulation and Real Time Applications (DS-RT 2000)*, San Francisco, California, USA, Aug. 24-26, 2000, pp. 97-102.

[PMODEL] Pro Model Corporation. Available via <http://www.promodel.com>.

[PRTI] The Aegis Technologies Group, Inc. Portable Runtime Infrastructure. Available via <http://www.aegistg.com/prti/prticut.html>.

[QUA97] Quaglia, Francesco, "Rollback-Based Parallel Discrete Event Simulation by Using Hybrid State Saving". In *Proceedings of the 9th European Simulation Symposium*, October 1997, pp. 275-279.

[QUA99] Quaglia, Francesco, "Combining Periodic and Probabilistic Checkpointing in Optimistic Simulation". In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, Atlanta, GA, USA, May 1-4, 1999, pp. 109-116.

[QUI03] Quinn, Michael J., "Parallel Programming in C with MPI and OpenMP". *McGraw-Hill*, June, 2003.

[REI02] Reichenthal, Steven, "The Simulation Reference Markup Language (SRML): A Foundation for Representing BOMs and Supporting Reuse". 02FSIW-038, In *Proceedings of 2002 Fall Simulation Interoperability Workshop*, Orlando, Florida, September 2002.

[RFB90] Reiher, Peter, R. M. Fujimoto, Steven Bellenot and David Jefferson, "Cancellation Strategies in Optimistic Execution Systems". In *Proceedings of the SCS Multiconference on Distributed Simulation*, January 1990, 22(1): 112-121.

[ROH03] Rohrer, M.W., "Maxmizing Simulation ROI with AutoMod". In *Proceedings of the 2003 Winter Simulation Conference*, New Orleans, LA, USA, Dec. 7-10, 2003, pp. 201-209.

[ROM02] Rohrer, Matthew W. and Ian W. McGregor, "Simulating Reality Using AutoMod". In *Proceedings of the 2002 Winter Simulation Conference*, San Diego,

California, USA, Dec. 8-11, 2002, pp. 173-181.

[RPH99] Price, Rochelle N. and Charles R. Harrell, "Simulation modeling and optimization using ProModel". In *Proceedings of the 1999 Winter Simulation Conference*, Phoenix, AZ, USA, Dec. 5 – 8, 1999, pp. 208-214.

[RTI02] The Defense Modeling and Simulation Office (DMSO), "High Level Architecture RTI-NG Programmer's Guide, Version 5". February 2002.

[RWJ89] Reiher, Peter L., Frederick Wieland and David Jefferson, "Limitation of Optimism in the Time Warp Operating System". In *Proceedings of the 1989 Winter Simulation Conference*, Washington D.C., USA, Dec. 4-6, 1989, pp. 765-770.

[RYT03] Ryde, M.D. and S.J.E. Taylor, "Issues Using COTS Simulation Software Packages for the Interoperation of Models". In *Proceedings of the 2003 Winter Simulation Conference*, New Orleans, Louisiana, USA, Dec. 7-10, pp. 772-777.

[SAM85] Samadi, B., "Distributed Simulation, Algorithms and Performance Analysis". Computer Science Department, PhD Thesis, University of California, Los Angeles, 1985.

[SAQ05a] Santoro, A. and F. Quaglia, "Transparent State Management for Optimistic Synchronization in the High Level Architecture". In *Proceedings of the 19th Workshop on Parallel and Distributed Simulation*, Monterey, CA, USA, June 2005, pp. 171-180.

[SAQ05b] Santoro, A. and F. Quaglia, "A Version of MASM Portable Across Different UNIX Systems and Different Hardware Architectures", In *Proceedings of 9th IEEE International Symposium on Distributed Simulation and Real Time Applications (DS-RT'05)*, Montreal, Canada, October 2005, pp. 35-42.

[SBW88] Sokol, Lisa M., Duke P. Briscoe and Alexis P. Wieland, "MTW: A Strategy for Scheduling Discrete Simulation Events for Concurrent Execution". In *Proceedings of the SCS Multiconference on Distributed Simulation*, February 1988, 19(3): 34-42.

[SCH00] Schmidt, B., "The Art of Modelling and Simulation: Introduction to the Simulation System Simplex3". *SCS-Europe Publishing House*, Ghent, 2000.

[SIMUL8] Simul8 Corporation 2005. Available via <http://www.simul8.com>.

[SRML] Simulation Reference Markup Language (SRML). Available via <http://www.w3.org/TR/2002/NOTE-SRML-20021218/>.

[SSK98] Straβburger, Steffen, Thomas Schulze, Ulrich Klein, and James O. Henriksen. "Internet-based Simulation using Off-the-Shelf Simulation Tools and HLA". In *Proceedings of the 1998 Winter Simulation Conference*, Washington DC, USA, Dec. 4-6, 1998, pp. 1669-1676.

[STA01] Stanley, Brian, "AutoMod: the AutoMode Product Suite Tutorial". In *Proceedings of the 2001 Winter Simulation Conference*, Arlington, VA, USA, Dec. 9-12, 2001.

[STE93] Steinman, J., "Incremental State Saving in SPEEDS Using C++". In *Proceedings of the 1993 Winter Simulation Conference*, Society for Computer Simulation, LA, CA, USA, Dec. 12-15, 1993, pp. 687-696.

[STR99] Straβburger, Steffen, "On the HLA-based Coupling of Simulation Tools". In *Proceedings of the 1999 European Simulation Multiconference*, SCS, Warsaw, Poland, 1999, pp. 45-51 (Vol.1).

[STR01] Straβburger, Steffen, "Distributed Simulation Based on the High Level Architecture in Civilian Application Domains". PhD Dissertation. April 2001.

[SWA03] Swain, J.J., "Simulation Reloaded: Sixth Biennial Survey of Discrete-Event Software Tools". *OR/MS Today*, 30:4, pp. 46-57, 2003.

[TAY01] Taylor, Simon J. E., "Reference Model Specification Version 1.0". Technical Report. *HLA-COTS Simulation Package Integration Forum (HLA-CSPIF)*, 2001.

[TAY02] Taylor, Simon J. E., "Interoperating COTS Simulation Modeling Packages: A Call for the Standardization of Entity Representation in the High Level Architecture Object Model Template". In *14th European Simulation Symposium and Exhibition*, Dresden, Germany, October 2002.

[TAY03] Taylor, S. J. E., "HLA-CSPIF: The High Level Architecture-COTS Simulation Package Interoperation Forum". In *Proc. Fall Simulation Interoperability Workshop*, Orlando, FL, 2003. 03F-SIW-126.

[TBF02] Taylor, S.J.E, A. Bruzzone, R.M. Fujimoto, B.P. Gan, S. Straβburger and R.J. Paul, "Distributed Simulation and Industry: Potentials and Pitfalls". In *Proceedings of the 2002 Winter Simulation Conference*, San Diego, USA, Dec. 8-11, 2002, pp. 688-694.

[TCG01] Turner, S.J., W. Cai and B.P. Gan, "Distributed Supply-chain Simulation Using High Level Architecture". *Transactions of the Society for Computer Simulation International*, Vol. 18, No. 2, Jun 2001, pp. 98-109.

[TOC63] Tocher, K.D., "The Art of Simulation". *English Universities Press*, London, 1963.

[TOD05] Tolk, A. and S.Y. Diallo, "Model-Based Data Engineering for Web Services", *IEEE Internet Computing*, 9:4, pp. 65-70, 2005.

[TTL05] Taylor, S.J.E., S.J. Turner and M.Y.H. Low, "The COTS Simulation Interoperability Product Development Group". In *Proceedings of the 2005 European Interoperability Workshop*. Simulation Interoperability Standards Organization, Institute for Simulation and Training, Florida, 2005, 05E-SIW-056.

[TTM05] Taylor, S.J.E., S.J. Turner, N. Mustafee, H. Ahlander and R. Ayani, "COTS Distributed Simulation: A Comparison of CMB and HLA Interoperability Approaches to Type I Interoperability Reference Model Problems". *SIMULATION*. 81, 1, 2005, pp. 33-43.

[TYT04] Turner, S.J., Y.Y. Yap, S.J.E. Taylor and M.Y.H. Low, "Comparing High Level Architecture Entity Transfer Mechanisms", Technical Report, Nanyang Technological University, 2004.

[UCC93] Unger, B. W., J. G. Cleary, A. Covington and D. West, "External State Management System for Optimistic Parallel Simulation". In *Proceedings of the 1993 Winter Simulation Conference*. Society for Computer Simulation, LA, CA, USA, Dec. 12-15, pp. 750-755.

[VAM99] Vardânega, Fernando and Carlos Maziero, "Simplifying Optimistic Distributed Simulation in the High Level Architecture". *Parallel and Distributed Computing and Systems*, Cambridge Massachusetts, USA, November 1999.

[VAM00] Vardânega, Fernando and Carlos Maziero, "A Generic Rollback Manager for Optimistic HLA Simulations". In *Proceedings of the 4th IEEE International Workshop on Distributed Simulation and Real-Time Applications (DS-RT'00)*, San Francisco, California, USA, August 2000, pp. 79-85.

[WBIM] IBM's product: Websphere Business Modeler. Available via <http://www-306.ibm.com/software/integration/wbimodeler/>.

[WES88] West, D., "Optimizing Time Warp: Lazy Rollback and Lazy Re-evaluation". Technical Report. *Computer Science Department, University of Calgary*, Alberta Canada, 1988.

Canada, 1988.

[WIE98] Wieland, F., "Parallel Simulation for Aviation Applications". In *Proceedings of the 1998 Winter Simulation Conference*, Washington, DC, Dec. 13-16, 1998, pp. 1191-1198.

[WIE99] Wieland, F., "The Threshold of Event Simultaneity", *Transactions of the Society for Computer Simulation International*, 1999, 16(1): 23-31.

[WITNESS] Lanner Group, Inc. Witness, available via <http://www.laner.com>.

[WSW91] Weatherly, R., D. Seidel and J. Weissan. "Aggregate Level Simulation Protocol". *Summer Computer Simulation Conference*, July 1991.

[XIC95] Xiao, Z., J. Cleary, et al., "A Fast Asynchronous Continuous GVT Algorithm for Shared Memory Multiprocessor Architectures". In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, Lake Placid, New York, USA, Jun. 13-16, 1995, pp. 203-208.

[XML] XML (Extensible Makeup Language) Standard. World Wide Web Consortium. Available via <http://www.w3.org>.

[XMSF] Extensible Modeling and Simulation Framework. Available via: <http://www.movesinstitute.org/xmsf/xmsf.html>.

[XPSL] Pipeline Simulation Interest Goup, XPSL (Extensible Pipeline Simulation Language) Plan. Available via <http://www.psig.org/Standards/PSIG%20001-2003.pdf>.

[ZBC99] Zeigler, B., G. Ball, H. Cho, J. Lee and H. Sarjoughian, "Implementation of the DEBVS Formalism over the HLA/RTI: Problems and Solutions". In *Proceedings of the 1999 Spring Simulation Interoperability Workshop*, Orlando, USA, March 1999, 99S-SIW-065.

[ZCL04] Zhou, Suiping, Wentong Cai, Bu-Sung Lee and Stephen J Turner, "Time-space consistency in Large-scale Distributed Virtual Environments". *ACM Transactions on Modeling and Computer Simulation* (TOMACS), Jan 2004, 14(1): pp.31-47.

[ZHA05] Zhang, J., "Interoperation Standards for Commercial Off-The-Shelf Simulation Packages". Final Year Project Report, Nanyang Technological University, 2005.