A SIMULATION PLATFORM FOR
EXPERIMENTATION AND EVALUATION OF
DISTRIBUTED-COMPUTING SYSTEMS

by

Yijia Xu

Copyright © Yijia Xu 2005

A Dissertation Submitted to the Faculty of the
DEPARTMENT OF SYSTEMS AND INDUSTRIAL ENGINEERING
In Partial Fulfillment of the Requirements

For the Degree of
DOCTOR OF PHILOSOPHY

In the Graduate College
THE UNIVERSITY OF ARIZONA

2005
THE UNIVERSITY OF ARIZONA ®  
GRADUATE COLLEGE

As members of the Final Examining Committee, we certify that we have read the dissertation prepared by Yijia Xu entitled A SIMULATION PLATFORM FOR EXPERIMENTATION AND EVALUATION OF DISTRIBUTED-COMPUTING SYSTEMS and recommended that it be accepted as fulfilling the dissertation requirement for the degree of Doctor of Philosophy.

_____________________________  __________________________
Suvrajeet Sen, Ph.D.  Date  05/06/2005

_____________________________  __________________________
Young Jun Son, Ph.D.  Date  05/06/2005

_____________________________  __________________________
Leonardo Lopes, Ph.D.  Date  05/06/2005

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

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

_____________________________  _________________
Suvrajeet Sen, Ph.D.  Date  05/06/2005

Dissertation Director
STATEMENT BY AUTHOR

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

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

SIGNED:  Yijia Xu
ACKNOWLEDGEMENTS

First of all, I would like to thank my advisor Dr. Suvrajeet Sen for his mentoring and support on my research in the Ph.D. program. His vision, insight, and methods to scientific research have guided me throughout the research and will continue to guide me in my professional career. I am very grateful to his continuous encouragement in my difficult times as well as his patience over the years to accomplish this research.

I thank Dr. Frank W. Ciarallo, Dr. Young Jun Son, Dr. Leo Lopes, Dr. Bernard P. Zeigler, and Dr. Saim A. Hariri, for their willingness to serve on my committee. Their comments, suggestions, and encouragements during the course of my Ph.D study are invaluable.

I would like to thank the department head Dr. Ronald G. Askin and all the members of staff in the SIE department, especially Linda Cramer, Meg Rosenquist, and Bill Ganoe. I also thank my colleagues in the RAPTOR lab for their help and encouragement: Lewis Ntaiou, Lei Zhao, Guglielmo Lulli, and Lihua Yu. I am grateful to ACIMS lab members Dr. Hessam S. Sarjoughian, Dr. Young Kwan Cho, and Dr. Xiaolin Hu for their generous helps. I am so indebted to Abihjit Dixit, A.L. Barath, Paul Thibodeaux, V. Vijay, Omrum Aki, and Ramish for their excellent job to set up the parallel reality experiments. We had a good time in the lab.

I appreciate Dr. Jim Byers at SunSet Software Technologies for the demonstration version of the XA linear programming solver.

This dissertation will not be possible without my wife Dan Chen’s love and support. Thank you for being with me all the time. I want to thank my parents Xing Xu and Ke Yang, as well as my sister Yike, for the encouragement and care.

This research was funded by the Next Generation Software Program (DMI 9978780 and CISE 9975050) of NSF.
DEDICATION

To my wife and my parents.
# TABLE OF CONTENTS

LIST OF FIGURES ................................................................................................................................. 9
LIST OF TABLES ....................................................................................................................................... 11

CHAPTER 1 INTRODUCTION ............................................................................................................. 14
  1.1 MOTIVATION ............................................................................................................................... 14
  1.2 DESIGN ISSUES .......................................................................................................................... 16
  1.3 ORGANIZATION OF THE DISSERTATION .................................................................................. 20

CHAPTER 2 BACKGROUND AND LITERATURE REVIEW .......................................................... 23
  2.1 DISTRIBUTED SIMULATION ....................................................................................................... 24
    2.1.1 Background ............................................................................................................................ 24
    2.1.2 Literature Review ................................................................................................................ 47
  2.2 AN OVERVIEW OF AGENT RESEARCH ...................................................................................... 50
    2.2.1 Background ............................................................................................................................ 50
    2.2.2 Software Agent Literature Review ....................................................................................... 52
  2.3 SUMMARY .................................................................................................................................... 55

CHAPTER 3 A DISTRIBUTED COMPUTING ARCHITECTURE FOR SIMULATION AND OPTIMIZATION ................................................................. 57
  3.1 INTRODUCTION ............................................................................................................................. 57
  3.2 BASELINE TECHNOLOGIES : AN OVERVIEW ......................................................................... 61
    3.2.1 Distributed Computing Environments ................................................................................... 62
    3.2.2 Decision Models .................................................................................................................... 63
    3.2.3 Network Middleware ............................................................................................................. 65
  3.3 SPEED-CS SYSTEM DESIGN ....................................................................................................... 67
    3.3.1 SPEED-CS Architecture ......................................................................................................... 68
    3.3.2 Network Programming .......................................................................................................... 71
    3.3.3 Modeling and Simulation Formalism ...................................................................................... 72
    3.3.4 A Common Data Model ....................................................................................................... 74
  3.4 IMPLEMENTATION ......................................................................................................................... 75
    3.4.1 Object Request Broker (ORB) layer ....................................................................................... 75
    3.4.2 Decision Model Components ............................................................................................... 77
    3.4.3 SPEED-CS Layer ................................................................................................................. 82
    3.4.4 Simulation Components ....................................................................................................... 86
    3.4.5 Integrated System ................................................................................................................. 87
  3.5 AN EXAMPLE .................................................................................................................................. 89
    3.5.1 The LP formulation ............................................................................................................... 90
    3.5.2 Transferring data from the LP to Simulation ....................................................................... 93
### TABLE OF CONTENTS — CONTINUED

3.5.3  The Simulator ........................................................................................................ 94  
3.5.4  Transferring data from Simulation to LP .............................................................. 96  
3.5.5  Simulation Results .............................................................................................. 98  
3.5.6  Class Diagram .................................................................................................... 100  
3.6  SUMMARY ............................................................................................................. 100  

#### CHAPTER 4  A MULTI-AGENT DATA COLLECTION ARCHITECTURE FOR DISTRIBUTED SIMULATIONS ................................................................. 103  
4.1  INTRODUCTION ..................................................................................................... 104  
4.2  A MAS FOR DISTRIBUTED DATA COLLECTION .................................................. 106  
4.2.1  Design Issues for Data Exchange Agent .............................................................. 106  
4.2.2  A Multi-Agent System ...................................................................................... 108  
4.2.3  The Data Exchange Agent Automaton ............................................................... 111  
4.2.4  Software Realization Using DEVSJAVA ............................................................ 116  
4.3  IMPLEMENTATION AND EXPERIMENTS .......................................................... 118  
4.3.1  Experimental setup ............................................................................................ 119  
4.3.2  Results and Analysis .......................................................................................... 125  
4.3.3  Class Diagram ................................................................................................... 130  
4.4  SUMMARY ............................................................................................................. 130  

#### CHAPTER 5  THE PARALLEL REALITY FRAMEWORK FOR DYNAMIC DATA DRIVEN SIMULATIONS ........................................................................... 134  
5.1  INTRODUCTION ..................................................................................................... 135  
5.2  PARALLEL REALITY FRAMEWORK ................................................................. 139  
5.2.1  Distributed Simulations and Dynamic Data Sources ........................................ 139  
5.2.2  Multi-Agent Based Systems ............................................................................. 140  
5.2.3  Functionality of Parallel Reality Framework ..................................................... 141  
5.2.4  Architecture of Parallel Reality Framework ..................................................... 143  
5.3  AN EXPERIMENTAL SYSTEM ...................................................................... 145  
5.3.1  Overview ........................................................................................................... 145  
5.3.2  Implementation of General-purpose Components ........................................... 147  
5.3.3  Application-Specific Implementation ................................................................. 151  
5.3.4  Data Processing and Parameter Updating ....................................................... 159  
5.3.5  Class Diagram ................................................................................................... 161  
5.4  EXPERIMENTAL RESULTS ........................................................................... 162  
5.5  SUMMARY ............................................................................................................. 165
TABLE OF CONTENTS — CONTINUED

CHAPTER 6  CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS ......167
  6.1 CONCLUSIONS ............................................................................................................................167
  6.2 FUTURE RESEARCH DIRECTIONS .............................................................................................169

APPENDIX A  SIMULATIONS WITH EXTENSIVE SUB-MODEL INTER-ACTIONS ........................................171

APPENDIX B  THE GRAPHIC USER INTERFACE OF PARALLEL REALITY SYSTEM ..........................................................175

REFERENCES ........................................................................................................................................179
LIST OF FIGURES

Figure 1.1 SPEED-CS modules...................................................................................20
Figure 3.1 SPEED-CS architecture ..........................................................................70
Figure 3.2 A linear programming interface ..............................................................78
Figure 3.3 An example of data stream to initialize an LP .........................................80
Figure 3.4 An example of objective function data stream ........................................81
Figure 3.5 Locating the generic LP object .................................................................84
Figure 3.6 A simulation model invokes a decision model .........................................87
Figure 3.7 A decision model invokes a simulation model .........................................88
Figure 3.8 The mapping for LP to simulator ............................................................89
Figure 3.9 Product-mix problem simulation .............................................................95
Figure 3.10 The mapping from simulation to LP ........................................................96
Figure 3.11 Average time data ..................................................................................97
Figure 3.12 Decision and simulation integration class diagram ................................102
Figure 4.1 Multi-agent system ..................................................................................110
Figure 4.2 Simulation setup ....................................................................................119
Figure 4.3 Experimental system .............................................................................122
Figure 4.4 Average total packets transferred between simulations .......................126
Figure 4.5 Average total traffic: agent system v. centralized approach ....................129
Figure 4.6 Average total traffic between computers ...............................................132
Figure 4.7 Multi-agent data collection system class diagram ..................................133
Figure 5.1 Parallel reality framework .......................................................................144
Figure 5.2 Parallel DEVS simulation protocol .........................................................148
Figure 5.3 Experimental system setup .................................................................152
Figure 5.4 An example of XML output schema from PathPlanner .......................153
Figure 5.5 An example of command information ....................................................154
Figure 5.6 Parallel reality class diagram ................................................................162
Figure 5.7 Travel time and delays ..........................................................................164
Figure A.1 Simulation snapshot: initial state .........................................................171
LIST OF FIGURES — CONTINUED

Figure A.2 Simulation snapshot: red truck to migrate. .............................................172
Figure A.3 Simulation snapshot: red truck finishes migration. ...............................173
Figure A.4 Simulation snapshot: red truck arrives at destination. ............................174
Figure B.1 The setup GUI .........................................................................................175
Figure B.2 Select path-planning algorithms ..............................................................175
Figure B.3 Select server hosts ..................................................................................176
Figure B.4 Add/Remove a server .............................................................................176
Figure B.5 Select real vehicles ................................................................................177
Figure B.6 Simulation controls ................................................................................178
Figure B.7 Simulation window..................................................................................178
LIST OF TABLES

Table 3.1 Initial expected processing time in minutes for LP ........................................90
Table 3.2 Expected processing time in minutes for simulation.....................................91
Table 3.3 Net profit of each product ............................................................................91
Table 3.4 Production plans with ±10.0% deviation from the average processing times ..........................................................99
Table 3.5 Production plans with ±20.0% deviation from the average processing times .........................................................................................99
Table 4.1 Data exchange agent’s “visiting interval” parameter and real time interval ........................................................................................................121
ABSTRACT

Distributed simulations have been widely applied as the method to study complex systems which are analytically intractable and numerically prohibitive to evaluate. However it is not a trivia task to develop distributed simulations. Besides distributed simulations may introduce difficulties for analysis due to decentralized, heterogeneous data sources. It is important to integrate these data sources seamlessly for analysis. In applications for system design, it is required to explore the alternatives of hardware components, algorithms, and simulation models. How to enable these operations conveniently is critical for the distributed system as well. All these challenges raise the need of a workbench that facilitates rapid composition, evaluation, modification and validation of components in a distributed system.

This dissertation proposes a platform for these challenges, which we refer to the SPEED-CS platform. The architecture of the platform consists of multiple layers that include network layer, component management layer, components layer, and modeling layer. It is a multi-agent system (MAS), containing static agents and mobile agents. The mobile agent is referred as the Data Exchange Agent, which is able to visit sub-simulations and has the intelligence to find the useful data for output analysis. Experiments show that the MAS requires much less network bandwidth than the “centralized” system does, in which simulations report data to output analyst.

The application of the SPEED-CS platform is extended to handle systems with dynamic data sources. We demonstrate that the platform can be used for parallel
reality applications where simulation parameters can be updated according to real-time sensor information. Data exchange agents are involved to manage the collection, dissemination, and analysis of data from dynamic data sources including simulations and/or physical systems.

The SPEED-CS platform is also implemented to integrate simulations and optimizations. The system is able to provide services to facilitate distributed computing, event services, naming services, and component management. One of the important features is that the component sets can be updated and enlarged with different models adding in. This feature enables the platform to work as a testbed to explore alternatives of system designs.

Finally we conclude this dissertation with several future research topics.
CHAPTER 1  INTRODUCTION

1.1 Motivation

Simulation has been a very important method for studying complex systems. A great number of applications of simulation have been reported in various fields such as network systems, military, manufacturing, financial study, education, video games, and many others. The common feature of these systems is that they are analytically intractable and numerically prohibitive to evaluate, which leaves simulation the only evaluation methodology. The objectives of the simulation include: 1) understanding the behavior of a system; 2) obtaining estimates of average performance measures; 3) guiding the selection of design parameters and/or operating strategies; and 4) fitting a simple performance model to measurements of the system (Righter and Walrand, 1989).

Generally the design of simulations for complex systems is difficult and time-consuming. Not only that, the running of simulation models is usually slow too. The complexity of the model, the limitation of the computer hardware, as well as the requirement of sufficient samples are the main reasons for the slow speed. Although the speed of a computer processor is continuously improving as the result of the research of the computer science and technology, it still requires a substantial amount of time to run and execute complicated simulations on one single computer. After all, the systems to study have no boundary, but the speed of a single computer has.
Therefore distributed simulation has received more and more attention in the simulation research community as a promising method to study the complex system problems and to increase the speed of simulations.

There are increasing needs for a distributed simulation platform, which can serve as architecture that facilitates rapid composition, evaluation, modification and validation of applications of distributed computing technology. In the following we list a few scenarios that such architecture applies.

Consider a collection of autonomous units equipped with sensor-based control systems which allow them to perform a set of tasks. These autonomous units may be heterogeneous, spatially distributed, and mobile. They may be endowed with their own local network to allow information exchange for coordinated control. Examples of such systems includes trucks and shovels in an mine, automated guided vehicles and loading/unloading devices on manufacturing floor, automated storage and retrieval systems in a warehouse, quay crane operations in major harbors (terminals), and so on. These autonomous units are also able to share information with each other and with one or more operations management units through the use of a different communication network. Thus, these systems involve components that perform a variety of functions which range from simple tasks (such as polling communications nodes) to more complex decision-oriented activities (such as system-wise work-load distribution).

There are often cases in which optimization models interact with simulation models. In some applications, simulations are used to generate scenarios which serve
as a sample space for stochastic decision models (such as stochastic programming).

In some other applications optimization models are designed to find the “best” system configurations via simulations. The interactions can also be seen in systems modeled with multiple fidelities, where tractability of an optimization model might require a coarse-grain stochastic model, while a detailed (fine-grain) simulation model may be used to examine the consequences of the solution provided by the optimization model.

This dissertation presents a Simulation Platform for Experimentation and Evaluation of Distributed-Computing System, in short the SPEED-CS framework. The goal which our research intends to achieve is a generic platform to study behaviors of aforementioned applications through the combinations of methods such as simulation, decision modeling, and physical experiments. To this end, the SPEED-CS framework is required to provide a variety of tools and interfaces that can be used to compose and modify these components. In addition, the framework shall not only provide rapid experimentation and evaluation, but also help monitor performance indices under various scenarios.

1.2 Design issues

The main research issue in developing the SPEED-CS framework involves the development of a methodology that integrates software across several architectural layers. Layered architecture provides encapsulation of components so that changes or upgrades of the components will have limited impact on the whole system. Layered
architecture can also provide increased reusability, dynamic extensibility, improved understandability, reduced development costs, and better supports for long-term system evolution (Coulson et al., 2002). The layers of the SPEED-CS architecture include network layer, component management layer, components layer, and user interface layer. The network layer is in charge of the communications over the network. The component management layer provides services to manage the components and services to coordinate simulation times. The components layer contains generic components or application-oriented components. User interface layer allows a user to access the components with an easy way.

Simulation applications built with the SPEED-CS framework can be decomposed to sub-models to be deployed over computer networks. Distributed computing system has the benefits such as speeding up the calculation and being able to model complicated systems, but it also has the complexity that each sub-model will generate substantial distributed data. This complexity raises the question of how to analyze the huge dataset in time. One of the traditional methods is to report the data to the main program for analysis. The disadvantage of this method is that it can generate a great amount of network traffic. Recently the method of using software agents to perform the data collection tasks has attracted a lot of attention. This approach can reduce the network traffic for large distributed simulation systems and provide good maintainability and scalability. The SPEED-CS framework is designed with a multi-agent system (MAS) which contains both stationary and static agents for data collection tasks.
In distributed simulation for complex systems, there are many data sources and the data sources are usually distributed too. Some simulations of complex systems are interoperable with real-time applications. Thus in this type of simulations, data are generated from sub-simulations and real-time applications. It is important for the simulation system to be able to integrate data from heterogeneous data sources. In some cases the real-time applications are real software code. For example, Zheng and Ni (2002) and Daniel and Kurt (2002) have simulated links among nodes in the target wireless network with regular computers and high-speed network devices while real codes are running at high layers. A module is developed between the network layer and device driver in each computer, to introduce effects such as propagation delay, packet loss or bandwidth throttling and to emulate dynamic network conditions. In other cases, the real-time data sources are hardware sensing systems. This type of interaction often arises in the form of *embedded systems*. The examples include digital control systems (Liu et al., 1999; Eker et al., 2001), high-performance data acquisition systems (Ludvig et al., 2002), embedded software-based self-testing for System-on-Chip (SoC) design (Krstic et al., 2002), and so on. These works only provide a tightly coupled system for a specific problem. They do not offer a generally practical framework for the integration between software world and physical world. We have approached this problem by designing and implementing the SPEED-CS framework as architecture to synthesize heterogeneous models. This research direction has drawn many interests and there are outcomes such as Ptolemy II (Neuendorf, 2004). Similar to the SPEED-CS framework, Ptolemy II can
describe models with different levels of granularity. While Ptolemy II focuses on the embedded control systems, the SPEED-CS platform is able to integrate decision models, simulation models, as well as physical systems. For instance, a forest fire prediction model can be integrated in the SPEED-CS framework with a forest fire simulation model that takes real-time fire situation data and wind information as inputs. Applications to explore dynamic systems consist of simulation and optimization models can be cleanly developed in the SPEED-CS framework.

System designs involve the explorations of the alternatives. As a simulation testbed, the SPEED-CS framework is required to provide researchers with easy methods to consider options of algorithms, hardware components, and simulation models. Figure 1.1 depicts the SPEED-CS modules, which allows users to section components to create an alternative applications in a “plug-and-play” manner. The users have the flexibility of exploring alternatives of the decision models and algorithms, and the solutions can be evaluated by running the simulation and/or the physical system. The user can also explore hardware options or different simulation models. The SPEED-CS Coordinator manages and integrates all these heterogeneous data sources. The system provides event services such that sensors, decisions, and simulations can invoke events in simulation models. On the other hand, events from simulation models can also invoke the running of decision models and new measurements. The architecture uses eXtensible Markup Language (XML) as a common data interface for the convenience of data exchange. With this protocol, the procedure of extracting information from data is standardized. Mobile Data
Exchange Agents are designed for retrieving data of interest for model parameter estimations, which avoid the situations where the sub-systems generate huge amount of network traffic to report data to the SPEED-CS Coordinator.

![Diagram of SPEED-CS modules](image)

**Figure 1.1** SPEED-CS modules

### 1.3 Organization of the Dissertation

This dissertation consists of six chapters and one appendix and is organized as follows. Chapter 2 presents the knowledge base of distributed simulations. The knowledge base covers general issues such as methods to decompose simulations and time management. Then we introduce the paradigms to implement a distributed simulation system. More recently federate paradigm is widely applied as it provides interoperability among simulations and reusability of simulation models. We introduce the High Level Architecture (HLA) specifications as the standard for federate paradigm. The Discrete Event System Specification (DEVS) is discussed as
a modeling and simulation formalism and the services that it can provide according to the HLA specifications are highlighted. We briefly review the researches of distributed simulation in manufacture control systems. Another important cornerstone of the SPEED-CS architecture is the research of software agent systems. We present an overview of the agent researches followed by a survey of the applications of software agents in distributed computing systems.

In Chapter 3 we demonstrate the integration of distributed simulation models and decision models within the SPEED-CS framework. The system is able to provide services to facilitate distributed computing, event services, naming services and component management. We also show the usages of XML as common data format for decision models and output data from components. The components are maintained in a component set in the SPEED-CS layer by their names and the component set can be enlarged by adding more model components. A product-mix problem is designed with both decision model and simulation to illustrate the applicability of the integrated system.

In Chapter 4 we propose a multi-agent system (MAS) for data collection and output analysis of a distributed simulation within the SPEED-CS framework. A distributed simulation is typically decomposed into sub-simulations that exchange information such as entity states, transitions, durations remaining in a certain state, and sometimes objects. These sub-models are data sources which generate a large amount of local data. In order to analyze the simulation, it is necessary to collect information from the distributed data sources. The traditional method is to report the
data to a centralized data source and then analysis is performed based on the centralized data sources. Although it is a straightforward approach, it has the disadvantage of generating too many network traffic. We develop a multi-agent system, which includes static agents to collect local data and mobile Data Exchange Agents to retrieve useful information. Experiments show that the MAS can reduce the network traffic significantly provided the data collection agent is able to choose its operational parameters appropriately.

Chapter 5 deals with the framework’s interoperability with real-time data source. The framework can be used for parallel reality system, in which real-time data are collected to ensure that the simulation system maintains the model fidelity. We use mobile agents presented in Chapter 4 to collect data. The framework is tested over a laboratory setup which we refer to as “open bit mine” to demonstrate the applicability.

Chapter 6 concludes the research covered in the dissertation and discusses the directions of future directions. The Appendix demonstrates two experiments: a distributed simulation of a mine system and an experiment of parallel reality.
CHAPTER 2  BACKGROUND AND
LITERATURE REVIEW

This chapter provides background of distributed simulations and agent systems, as well as a survey of related researches in these two fields. Distributed simulation relies on parallel computing systems for simulations and is sometimes referred as parallel and distributed simulation. We review simulations and then describe different ways to decompose a discrete-event simulation. We also discuss the time management of distributed simulations. Then we review the paradigms for implementing a distributed simulation system. Because of its interoperability among simulations and reusability of simulation models federate paradigm has become one of the most popular paradigms in the recent research studies. Thus as the standard for federate paradigm the High Level Architecture (HLA) specifications is introduced. In the implementation of the SPEED-CS project, we apply DEVSJAVA as the modeling and simulation tool set. DEVSJAVA follows the Discrete Event System Specifications (DEVS), which provides very similar services as HLA for distributed computing systems. Another cornerstone in the implementation of the SPEED-CS architecture is software agent. In this chapter we give a review of the software agent researches as well as a brief survey of the applications in distributed computing systems.
2.1 Distributed Simulation

2.1.1 Background

2.1.1.1 An Overview of Simulation

Simulation models are developed as representations with certain level of abstraction of physical systems to be studied. A physical system is described in a simulation with states and times among other parameters. If the change of a state is instantaneous and occurs at discrete points in simulation time, the simulation is a Discrete Event Simulation; if the state changes continuously over time, the simulation is referred to as Continuous Simulation (Law and Kelton, 1999).

In discrete event simulations, there are two approaches to advance the simulation time. One approach is to increase the simulation time by a constant time step \( \Delta \). Simulations with this type of time advancing approach (unit time approach) are called time-driven discrete simulations. In the other approach, the increment of simulation time is triggered by the next earliest occurring event. The second approach is referred to as event-driven approach. It is well known that event-driven approach is a better choice for general-purpose simulations, where events are often generated randomly (Ferscha, 1995).

For a long period of time, discrete event simulations run sequentially. The set of future events is maintained in a global event list and an event scheduler schedules the event in a non-decreasing timestamp order. After one event is processed, the simulation clock recording the current (simulated) time is then advanced to the time
of the next earliest occurring event. The sequential simulation cannot satisfy the requirements of faster simulation for complex systems. One obvious solution for obtaining a faster simulator is to decompose the simulation and to distribute the tasks on more resources. Prior to the discussion of different implementation methods of decomposing a simulation, in the next section we first review the computer network architectures that are available for distributed simulations.

2.1.1.2 Distributed Computing Architectures

There are mainly two architectures available for distributed computing systems. One is distributed shared memory system, and the other one is message-passing system.

A distributed shared memory (DSM) system has multiple processors but a common shared memory that all processors can access. The BBN Butterfly system (BBN Laboratories, 1986), the Connection Machine (Hillis, 1985), and the sequent system are examples of shared memory machines. A shared memory system provides processors in a system with shared address space. Application programs can use this space in the same way as they use normal local memory. That is, data stored in the shared space are accessed through “read” and “write” operations. As the result, applications can pass shared information by reference. The major advantage of a shared memory system is that it simplifies the programming task.

In a message-passing system, communication among processors is achieved by message passing on a communication network. The communications are realized via
“send” and “receive” operations among the processors. Examples of message passing systems include Hypercubes, Transputer, and other systems that are more commonly used in business and university laboratories such as network of PCs or Sun workstations. Many of the message-passing systems can be deployed with certain middleware to accomplish detailed communication, which can relieve some programming hassle.

Both shared memory architectures and message-passing architectures can be used for distributed simulation. Depending on the type of the application, the method of decomposition, the synchronization methods, and other conditions, one or the other may be chosen for the distributed simulation. In some cases a hybrid architecture that combines these two architectures may be more appropriate and thus be used. The distributed network of shared memory machines is such an example.

2.1.1.3 Simulation Decompositions

Various ways to decompose a simulation motivate different implementations of the distributed simulation, as summarized below (Righter and Walrand, 1989; Vee and Hsu, 1999):

1. Parallelizing Compilers

This approach applies a parallelizing compiler to a sequential simulation program. Such compilers try to find sequences of the code that can be processed in parallel and schedule them on separate processors. This approach is transparent to the user and can be applied to the large body of existing sequential simulation software. The
disadvantage of this approach is that the obtained parallelism is limited because the compiler only considers the sequences of the code but completely ignores the structure of the problem.

2. Replicated Trials

The “Replicated Trials” approach is also called Parallel Independent Runs or Distributed Experiments. It is designed for N number of sequential simulations running independently on N processors for final output analysis. This approach is efficient to do independent replications of the simulation on different processors. Heidelberger (1986) has developed conditions on the bias and variance of estimates of steady state performance measures obtained from simulations under which the distributed experiments approach is more efficient than other approaches in terms of minimizing the mean squared errors of the estimates. Another application of this approach is to do independent runs with different parameter values simultaneously. One of the drawbacks of “Replicated Trials” approach is its lack of considering and using the interactions among simulations. Hence it cannot be used in situations in which parameters for each run depends on the results of previous run. It also cannot be applied in simulations where coordination is necessary among simulators. For each processor the single run may still be time-consuming and may require large amount of memory. These drawbacks limit generic applications of this approach.
3. Distributed Functions

The “Distributed Functions” approach decomposes a simulation to a set of subtasks and assigns the obtained subtasks to a number of processors. One processor can be dedicated to one or more of the following functions: random variable generation, event set processing, statistics collection, I/O, file manipulation, graphics generation, supervision, etc. The advantage of this approach is that it avoids deadlock problems and that it requires minimum changes in the code for sequential simulation. Its disadvantage is that it does not exploit any of the parallelism in the system being modeled.

4. Distributed Events (with centralized Event List)

This approach maintains a centralized event list as in traditional sequential simulation. When a processor becomes available, it continues to process the event with the earliest timestamp in the centralized event list. Protocols to preserve consistency are required, since the next event on the list may be affected by events currently being processed. This approach is particularly appropriate for shared memory systems, in which the event list can be accessed by all processors. On the other hand it requires knowledge of the simulation model, which may not be easily obtained. Furthermore the centralized event list can become a bottleneck if many processors are involved in the simulation and trying to access to the centralized event list simultaneously.
5. Domain Decomposition

This class of decomposition is based on the view that a simulation execution is equivalent to filling in a two-dimensional region: one dimension represents the simulated time and another dimension represents the state variables. According to this conception, we can design two types of decompositions: Time-Parallel Decomposition and Space-Parallel Decomposition (or Distributed Simulation). Time-Parallel Decomposition partitions the time domain into a number of disjoint intervals and each processor is assigned with one interval and carries out the simulation during the assigned interval. This decomposition requires that adjacent time intervals satisfy the initial states condition. The basic condition is that the resulting states from the simulation during the earlier time interval should be the initial states for the simulation in the next time interval. While there are arising interests of research on this approach, Space-Parallel Decomposition is a more mature approach. In Space-Parallel Decomposition approach, the simulation model is decomposed into a number of sub-models or components in space domain. Each component is assigned to one process, where several processes may be run on the same processor. The components exchange information between each other based on the coupling relations. The event list is also decomposed into individual ones to ensure thorough parallelism.

2.1.1.4 Time Management

Time management is a basic service in parallel and distributed simulations. It ensures that the simulation time in each of the simulator instances stays synchronized
with others. There are two widely-used methods to deal with the requirement for
time synchronization: conservative method and optimistic method.

1. Conservative Mechanisms

Conservative mechanisms require that messages from any simulator to any other
simulator be transmitted in chronological order according to their timestamps. In this
method, the simulators must have some means to determine when messages are *safe*
to send. A message is *safe* when it can be determined that no message will be later
received with an earlier timestamp.

With the conservative synchronization, a simulator must block an unsafe event.
This may lead to a deadlock situation if the algorithm is not properly designed. Next,
we use an example to demonstrate how the following simple algorithm can run into
the deadlock situation. Assume that the sequence of timestamps on the messages sent
over a link is non-decreasing. There is a message queue and a clock associated with
each link. For a particular link, if the queue is empty, then its clock is normally set to
the timestamp of the last received message; otherwise, the clock is set to the
timestamp of the message in the front of the queue. A simulator selects the link with
the smallest clock value, and sends the first message in the front of the link’s queue.
If the queue is empty, then the simulator blocks; it resumes when the queue of the
link with the smallest clock value becomes nonempty. It can be proven that this
algorithm adheres to the local causality constraint, and therefore no causality error
will ever occur (Chandy et al., 1979, Misra, 1986). However it is possible that some
simulators become blocked and each of them waits indefinitely for each other in a cyclic fashion. This indefinite loop can then lead to deadlock.

- **Deadlock Avoidance**

  One of the first methods developed to avoid deadlock, known as the CMB approach, was proposed independently by Chandy and Mirsa (1979) and Bryant (1977). The CMB approach uses a *null message* \((null, T_{null})\) to allow each simulator to make a local decision regarding the lower bound of timestamps (LBTS). A simulator determines the timestamp of a null message that it plans to send by checking the clock value of each incoming link and the simulated time which will elapse when processing an event. Whenever a simulator finishes processing an event, it can send null messages to all simulators to whom it links. It has been shown that this approach is able to avoid the deadlock situation as long as there is no cycle of zero timestamp increment. However it may generate a large number of null messages that would slow down the communication as well as simulation.

- **Lookahead**

  One way to reduce the traffic generated by null messages is the *lookahead* method, which allows the simulator to predict what will happen in the next cycle. Chandy and Misra (1981) define “lookahead” to be the amount of time that a process can look into the future. If the local clock of the simulator is at time \(t\), and the simulator can predict all messages it will send with time stamps less than \(t + \Delta\), then \(\Delta\) is the lookahead. Plenty of research has been done to utilize and improve the
lookahead approach. A scheme that can improve the lookahead capability by pre-
computing the service times for some future events is proposed by Nicol (1988).
Wagner and Lazowska (1989) have derived the expressions for the lookahead for
certain queuing network simulation. Furthermore, Cai and Turner (1990) have
introduced carrier null message, which play a similar role as that of the null messages,
but carry additional information on lookahead and the route taken. A framework for
automatic lookahead computation was suggested by Cota and Sargent (1990). In
their framework, they represent process behavior with control flow graph and show
how lookahead information can be deduced from the graph.

- Deadlock Detection and Recovery

An alternative to the lookahead method has been proposed by Chandy et al.
(1981) and Misra (1986). In their method, a simulation proceeds until deadlock, and
then deadlock is detected and corrected by the algorithm. Chandy et al. (1981) and
Misra (1986) involve a “controller” process to monitor for deadlock and control
deadlock recovery. In a shared memory system, flags or counters in global memory
can be used for deadlock detection (Reed et al., 1988, Fujimoto, 1988).

- Time Windows

Another research direction for the conservative time management is the time
windows method. This method defines some time intervals to determine safe events.
Lubachevsky (1989) proposed an algorithm with three elements: 1) the bounded lag,
which means that the difference in the simulated time between events being processed
concurrently is bounded by a known finite constant; 2) *minimum propagation* delays between the pair of the simulated system; and 3) the so-called *opaque periods*, which are the delays caused by the non-preemptive states that these parts can enter. Ayani and Rajaei (1992) conducted extensive experimental measurements on feed-forward networks and networks with feedback loops. They have concluded that the conservative time window (CTW) algorithm produces good speedup for symmetric workloads, especially for cases where size of a network, message population, and the amount of computation time needed to process each event are large. The main drawback of this method is that the time window has to be decided with specific knowledge of the simulation.

- **Conditional Events**

  Chandy and Sherman (1989) proposed a conditional event approach to determine a range of safe events by finding a minimum of all possible events from any peer. Events are classified into two categories: *definite events* and *conditional events*. Definite events, which generally have smaller timestamps, will definitely be processed and will not be cancelled or disabled by any other events. Conditional events, on the other hand, will be processed only when certain criteria are satisfied. In this way, the event with the smallest timestamp in the system is always a definite event. This approach requires the conditional knowledge of the events, which is used to distinguish the definite events and conditional events.
2. Optimistic Mechanisms

Using optimistic method, simulators will process events even with no guarantee that they will be processed in timestamp order. If an event is processed out of order, a causality error occurs.

A causality error is detected whenever a simulator with local clock $Clock_i$ receives an event (called a straggler) with a timestamp $T_{str} < Clock_i$. The simulator has to undo the effect of the out-of-order event, and return the state of the simulator to what existed prior to processing the straggler. This process is called a *rollback* (Vee and Hsu, 1999).

- **Time Warp**

  The basic optimistic approach is called Time Warp, which was proposed by Sowizral and Jefferson (1985; Jefferson, 1985). Under the Time Warp algorithm, each message has two time stamps, a send time and a receive time. The receive time is the simulated time of arrival at the receiver. The send time is the local clock of the sender when the message was sent, and is required for implementation of the Time Warp algorithm. The original Time Warp algorithm uses aggressive cancellation, where a simulator that rolls back to simulated time $T$ immediately sends out anti-message for any previously sent messages with timestamps greater than $T$. There will also be propagations of the sending of anti-messages, because the straggler may have caused chained effects on coupled simulators.
• Aggressive Cancellation Improvements

The aggressive cancellation can be expensive in terms of network traffic and computational time, because the frequent rollback operations generate a large amount of anti-messages and many computations have to be reprocessed. In attempt to resolve these problems, Gafni (1988) proposed a lazy cancellation method. In lazy cancellation, the process does not send anti-message immediately when a causality error occurs. Instead, the process resumes executing forward in simulation time from its new local virtual time (LVT), and when it produces a message, it compares the message with the one in its output queue. Only messages that are different from previously sent messages are transmitted, and only anti-messages that are not reproduced in the forward computation are transmitted.

Another improvement to the aggressive cancellation is lazy reevaluation (also called jump forward). The approach is similar to lazy cancellation, but deals with state vectors rather than messages (West, 1988). If the state of a process is the same after processing a straggler event message as it was before, then there is no need to re-execute the rollback events. This approach requires a comparison of state vectors to determine whether the state has changed, therefore it needs additional storage and bookkeeping overhead which can significantly complicate the Time Warp code (Fujimoto, 1990).

• Time Windows
Similar to conservative mechanism, window-based approaches have been proposed for optimistic mechanisms with the goal of reducing the number of causality errors. The Moving Time Window (MTW) approach uses a fixed time window of size $W$ (Sokol et al., 1988). Only events with timestamps in the interval $[T, T+W]$, where $T$ is the smallest time stamped event in the simulation, are eligible for processing. The benefit of time window approach is still arguable. Critics have pointed out that the windows cannot distinguish good computations from bad ones, so they may impede the progress of correct computations. Furthermore the correct computations beyond the upper bound of the window cannot be processed as well. Finally, it is not clear how to determine the size of the window. Lubachevsky, Shawrtz and Weiss (1989) have proposed filtered rollback which is a combination of the Time Warp and the bounded lag simulation algorithms. The algorithm contains tunable parameters which at one extreme make it identical to the bounded lag algorithm, and at the other extreme make it identical to the MTW approach.

- Wolf Calls

Madisetti, Walrand, and Messerschmitt (1988) have proposed the Wolf Calls scheme to prevent the erroneous computation from spreading widely. Processes that may be “infected” by the erroneous computation are notified when a straggler message is detected. This approach however may cause some correct computations to be frozen unnecessarily. It requires that one know the speed in real-time at which both the erroneous computation can spread, and the time required to transmit the
control messages. It can be difficult to determine bounds on these quantities for some simulation models.

2.1.1.5 Distributed Simulation Paradigms

There are two principle paradigms for constructing parallel and distributed simulations (Ferenci et al., 2000), the specific tool paradigm and federate paradigm. The specific tool paradigm defines a parallel simulation engine, associated languages, libraries, and tools to create new high performance simulators. The federate paradigm utilizes runtime infrastructure (RTI) to interconnect simulators that may be developed by different parties with different tools.

1. Specific tool paradigm

Specific tool paradigm is the traditional approach to realize parallel network simulator, which creates a new system “from scratch”. Examples of this paradigm include TéD (Perumalla et al., 1998), SPEEDES (Steinman et al., 2003), and Task-Kit (Xiao et al., 1999). In the following we briefly introduce TéD to illustrate the features of this paradigm.

TéD is a language designed mainly for modeling telecommunication networks. The TéD language specification is composed of two distinct parts – MetaTéD and “external language”. MetaTéD is a framework that defines a set of concepts for modeling the dynamic interactions of entities and their compositions in an application-independent manner. The “external language” can be regular general-purpose programming language, such as C++. 
The TeD abstractions include entity, architecture, and process (Bhatt et al., 1998). The physical and conceptual objects in the telecommunication domain are modeled in terms of entity descriptions. Entities are connected to and interact with each other via channels, which is the only means of dynamic interaction between entities. Channels transmit the events that describe dynamic interactions between entities.

TeD uses architecture to describe the dynamic behavior of each entity. The architecture of an entity is expressed using concurrent process semantics. It describes the actions of the entity upon event arrivals on its input channels.

Processes are dynamic threads of computation acting on behalf of the entities that own them. The functionality of processes consists of combinations of two types of actions: computation and synchronization. Computation is a sequence of operations performed on the state variables. Synchronization is the sequence of actions involving events on the channels or a lapse of time.

TeD models are complied into C++ code that can directly use the services of the parallel simulation kernel. Both optimistic and conservative time management mechanisms have been implemented as TeD targets (GTW and Nops).

While the specific tool paradigm has the advantage that the software can be tailored for parallel execution, a main drawback is that it is difficult and time consuming to develop such a tool. There are also substantial verification and validation efforts required, but the reusability of the software is limited.
2. Federate paradigm

The second paradigm is to federate disparate simulators, utilizing runtime infrastructure (RTI) to interconnect them. This method appears in many works such as Distributed Interactive Simulation (DIS) (Hofer and Loper, 1995), and Aggregate Level Simulation Protocol (ALSP) (Wilson and Weatherly, 1994), and the High Level Architecture (HLA) (US Department of Defense, 1998). The HLA seeks to generalize and build upon the research results from the DIS and ALSP. It has become the standard technical architecture supporting reuse and interoperation across the Department of Defense (DoD) simulation models. The HLA has been accepted as IEEE standard (IEEE 1516) supported by the Simulation Interoperability Standards Organization (SISO).

The HLA supports component-based simulation development, where components are individual simulations. Federations are composed of these simulations with well-defined functionality and interfaces. A federation is the combination of a particular federation object model (FOM), a set of federates, and the run-time infrastructure (RTI) services. Federates include simulation utilities, simulations, and live player interfaces. The RTI is a distributed operating system for the federation (Buss and Jackson, 1998). Note that the HLA defines an architecture rather than an implementation (Khul et al., 1999). The HLA contains three main components: the HLA rules, the HLA interface specification, and HLA object model template (OMT).
A. HLA Rules

The HLA rules express design goals and constraints of HLA-compliant federates and federations. There are five federation rules and five federate rules (US Department of Defense, 1998).

Federation Rules:

1. Federations shall have FOM in OMT format.

2. All representations of objects shall be in the federates and not the RTI.

3. During federation execution, all exchange of FOM data shall via the RTI.

4. During federation execution, all federates shall interact with the RTI in accordance with the interface specification.

5. During federation execution, an attribute of an instance of an object may be owned by only one federate at a given time.

Federate Rules:

6. Federates shall have a Simulation Object Model (SOM) in OMT format.

7. Federates shall be able to update/reflect attributes and send/receive data in accordance with their SOM.

8. Federates shall be able to transfer/accept attribute ownership in accordance with their SOM.

9. Federates shall be able to vary the conditions under which they provide attribute updates in accordance with their SOM.
(10) Federates shall be able to manage local time in a way which will allow them to coordinate data exchange with other members of the federation.

B. HLA interface Specification

The interface specification is a standard for federates to interact with the RTI (US Department of Defense, 1998). It defines how to provide the RTI services. The interface specifications have six basic RTI service groups: federation management, declaration management, object management, ownership management, time management, and data distribution management.

- Federate Management provides services for creation, dynamic control, modification, and deletion of a federation execution.

- Declaration Management provides services to declare their intent to publish and subscribe to object attributes and interactions.

- Object Management provides services to deal with registration, modification and deletion of object instances and the sending and receipt of object interactions.

- Ownership Management provides services to transfer ownership of instance attributes.

- Time Management provides services to coordinate the advance of logical time and maintain its relationship to real time.

- Data Distribution Management (DDM) provides services to reduce the transmission and receipt of irrelevant data.
C. HLA Object Model Template

The Object Model Template (OMT) is a common method for prescribing the information contained in the HLA object model for each federation and simulation (US Department of Defense, 1996). The HLA contains two types of object models: Federation Object Models (FOMs) and Simulation Object Model (SOMs) (Buss and Jackson, 1998).

Each of the object models is designed to provide certain services. The FOM is to provide a common specification for exchange of data and general coordination among members of a federation. The SOM is to provide a common, standardized mechanism for describing the capabilities of potential federation members. Three components in the OMT are critical to this design: (1) object classes, (2) interaction classes, and (3) routing spaces and their dimensions. Object and interaction classes represent the data that can be exchanged among federates. Objects classes contain attributes and interaction classes contain parameters. An object class has a lifetime associated with it. An interaction class (event) can only exit instantaneously. The other component is to facilitate efficient data distribution (routing) among federates. (Zeigler and Sarjoughian, 2002).

2.1.1.6 DEVS– A Federate M&S Framework

In this section, a modeling and simulation (M&S) framework – DEVS framework will be introduced. The DEVS (Discrete Event System Specification) formalism can support hierarchical modular model construction and component reuse. It is shown to
possess the six management services (ie. federation, object, data, declaration, ownership, and time) that are suggested by HLA. In the research of the SPEED-CS project, we apply the DEVS modeling and simulation formalism to develop components.

1. Discrete Event System Specification

The Discrete Event System Specification (DEVS) formalism provides a means of specifying a mathematical object called a system. Basically, a system has a time base, inputs, states, and outputs, and functions for determining next states and outputs given current states and inputs (Zeigler and Sarjoughian, 2003a).

A basic DEVS is a structure:

\[
DEVS = (X_M, Y_M, S, \delta_{cut}, \delta_{int}, \delta_{con}, \lambda, \tau)
\]

where

\[
X_M = \{(p, v) \mid p \in \text{InPorts}, v \in X_p\}
\]

is the set of input ports and values;

\[
Y_M = \{(p, v) \mid p \in \text{OutPorts}, v \in X_p\}
\]

is the set of output ports and values;

\[S\]

is the set of sequential states;

\[
\delta_{cut} : Q \times X_M^b \rightarrow S
\]

is the external state transition function;

\[
\delta_{int} : S \rightarrow S
\]

is the internal state transition function;

\[
\delta_{con} : Q \times X_M^b \rightarrow S
\]

is the confluent transition function;
\[ \lambda : S \rightarrow Y^b \] is the output function;

\[ ta : S \rightarrow R_0^+ \cup \infty \] is the time advance function;

with \( Q \equiv \{(s,e) \mid s \in S, 0 \leq e \leq ta(s)\} \) the total set of states.

A coupled model can be represented as

\[ CM =< D, \{M_i\}, \{N_i\}, \{Z_{i,j}\} > \]

where

\( D \) is the set of component names;

\( M_i \) is the set of basic components for each \( i \) in \( D \);

\( N_i \) is the set of influences for each \( i \) in \( D \);

\( Z_{i,j} \) is \( i \) to \( j \) output translation for each \( i \) in \( D \) and each \( j \) in \( N \).

There are three types of coupling: internal coupling, external input coupling, and external output coupling. Internal coupling interconnects the components of a coupled model. External input coupling interconnects the input ports of a coupled model to the input ports of its components. External output coupling interconnects the component output ports of a coupled model to the output ports of the coupled model itself. The port provides a generic pipe, through which objects can be sent and/or received.
2. DEVS/HLA

DEVS uses the system entity structure (SES) formalism to provide an operational scheme for specifying hierarchical structure. An SES is a structural knowledge representation scheme that systematically organizes a family of possible structures of a system. Such a family characterizes decomposition, coupling, and taxonomic relationships among entities (Zeigler and Sarjoughian, 2002). An entity is a representation of a real-world object.

DEVS formalism is able to provide the services that mentioned in HLA introductions. The DEVS and HLA can be seen as a complimentary relationship, which achieves high interoperability and reusability for distributed modeling and simulation. In the following section, we discuss the DEVS formalism with respect to the HLA services to provide (Zeigler and Sarjoughian, 2003b).

In the DEVS formalism, the atomic model represents dynamic behavior of an entity or sub-entity, which has the same functionality as the simulation object model (SOM) in HLA. The coupled model can represent distributed model, which is the functionality of federation object model (FOM). DEVS coupled model can be either a federation or a federate. A federation in DEVS can be composed of coupled models. DEVS provides federation management service with its modeling features, which not only supports interface modeling but also specifies behavioral dynamics. This kind of modeling feature is referred to *Dynamic Modeling*.
The declaration management in DEVS is accomplished with the concepts of ports and couplings. The information flow is realized with message passing from output ports to input ports whose connectivity is specified by couplings.

DEVS supports object management services with a different and complementary perspective from the HLA specifications. DEVS supports data exchange via ports/coupling among components which is represented in a coupled model, whereas HLA supports data exchange as either updates and/or interactions via an intermediary known as FED file (US Department of Defense, 1998).

DEVS currently does not support ownership management. There are ongoing research efforts which investigate a formal ownership representation for atomic and hierarchical coupled models to support the usual connotations of ownership including control of execution, sharing and property rights.

DEVS uses Predictive Contracts (Zeigler, Sarjoughian, and Praehofer, 2000) to accomplish the data distribution management, whereas HLA uses Routing Space (US Department of Defense, 1998). The routing space enables “as needed” data exchange based on federates’ proximity to one another in routing space, attribute updates, and parameter interactions. The predictive contract concept is applicable to both discrete and continuous systems. It provides additional filtering based on the content of data as well as spatial encounter prediction techniques (Zeigler, Hall, and Sarjoughian, 1999) to minimize unnecessary data exchange.
DEVS supports distributed simulation with a centralized time management scheme, which is similar to that of HLA. There are still differences in approaches though. HLA relies on time stamps assigned to messages and ensures adherence to a very strict causality principle of physical systems. The parallel DEVS protocol is based on message passing and does not require the messages to be time stamped. Furthermore, DEVS offers the unique capability of generating outputs in zero simulation time. The parallel DEVS protocol provides a basis for developing customized time management schemes.

2.1.2 Literature Review

Distributed simulations have been widely applied in manufacturing simulation and control systems. The needs to reduce manufacturing lead-time, and meet customer specifications without increasing manufacturing costs have led manufacturers towards “Enterprise Computing”. The notion of Enterprise Computing (EC) includes Factory Control Systems, Enterprise Resource Planning (ERP) and Supply Chain Management (SCM).

The design and implementation of factory control systems involves the ability to exchange design data from CAD workstations to manufacturing process planning computers, which in turn communicate with manufacturing equipment to specify optimal process parameters. In addition, the factory floor consists of loading/unloading devices, automated guided vehicles and inventory tracking systems.
which report on the status of work-in-process inventory, inspection results, units for re-work, and so on.


There have been other researches towards integrating the simulation model of a manufacturing system with the shop-floor control system. In these researches simulations can be used as tool for real-time evaluation in many circumstances. Davis and Jones (1988) first proposed the benefits of simulation for the real time and decision control of a manufacturing system. Kim and Kim (1994) extended their work by building a model in which dispatching rules were dynamically varied based on evaluation of the candidate rules. Drake, Smith and Peters (1995) presented a framework for applying simulation models to on-line planning, scheduling, and control problems. Son et al. (1999) presented a multi-pass simulation-based, real-time scheduling and shop floor control system. A multi-pass simulation is composed of a real-time simulation and a preview simulation, which runs in the fast mode. The real-time simulation acts as a real-time shop floor controller, whereas the fast-mode simulation acts as a real-time scheduler.
The applications of distributed simulation are not limited to shop floor controls; in areas such as supply chain systems interactions between decision algorithms and distributed simulations are important as well. Hung and Leachman (1996) introduced an ad-hoc system for automated production planning of semiconductor manufacturing based on iterative linear programming optimization and discrete-event simulation calculations. Godding, Sarjoughian and Kempf (2004) described a multi-formalism modeling approach for the design and operation of semiconductor supply/demand networks. The approach supports the modeling and interoperability of a semiconductor supply/demand network with discrete event simulation and linear programming.

In spite of significant research of the applications of distributed simulation and control systems, little research has been done to design and develop a generic framework to automatically analyze and utilize information collection from heterogeneous data sources. The data sources include decision module, simulation module, and physical reality module. There is lack of generic methodology of connecting these modules. In order to address these issues, we propose a multi-agent system for intelligent data collections within the SPEED-CS architecture. Furthermore the architecture helps set up connections among different modules. These connections allow the integrated system to dynamically adjust the decision or control based on the data analysis.
2.2 An Overview of Agent Research

2.2.1 Background

The agent research can find the root in robotics, as many consider an agent to be a “soft robot” living and doing its business within the computer world (Kay, 1990). Nwana (1996) splits agent research into two main strands: the first beginning about 1977, and the second around 1990. Strand one is closely related with distributed artificial intelligence (DAI) and has concentrated mainly on deliberative type agents with symbolic internal models. Strand two is a recent, rapidly growing movement to study a much broader range of agent types, emphasizing agent’s ability to accomplish tasks (Bradshaw, 1997).

There have been many attempts to define a software agent. Nowadays most agent researchers find the following definition acceptable: a software agent is a software entity which functions continuously and autonomously in a particular environment, often inhabited by other agents and processes (Shoham, 1997). Each agent may possess to a greater or lesser degree attributes like the ones enumerated in Etzioni and Weld (1995) and Franklin and Graesser (1996):

- *Reactivity:* the ability to selectively sense and act
- *Autonomy:* goal-directedness, proactive and self-starting behavior
- *Collaborative behavior:* can work in connect with other agents to achieve a common goal
• “Knowledge-level” (Newell, 1982) communication ability: the ability to communicate with persons and other agents with language more resembling human like “speech acts” than typical symbol-level program-to-programming protocols

• *Inferential capability*: can act on abstract task specification using prior knowledge of general goals and preferred methods to achieve flexibility; goes beyond the information given, and may have explicit models of self, user, situation, and/or other agents

• *Temporal continuity*: persistence of identity and state over long period of time

• *Personality*: the capability of manifesting the attributes of a “believable” character such as emotion

• *Adaptivity*: being able to learn and improve with experience

• *Mobility*: being able to migrate in a self-directed way from one host platform to another.

Many types of agents appear in specific applications. They can be classified into regulation agents, planning agents, or adaptive agents. A regulation agent is able to react to each sensory input and knows what to do, whereas planning agents have the ability to look ahead. A planning agent could be subdivided into more specific agents based on the way they plan. For example, a problem-solving agent may have some AI sense, case-based agents make decisions according to cases, OR agents use operation research methods, and so on. Adaptive agents have even more intelligence
because it not only plans, but learns as well. There are many different classifications of agents. Franklin and Graesser (1996) give a tree structure of agents in which software agents include task-specific agents (like Sumpy), entertainment agents (like Julia) and computer viruses. Maes (1994) developed personal or interface agents helping people reduce workload and information load. A personal assistant could have the ability of handling email, scheduling meetings, filtering news, and so on.

2.2.2 Software Agent Literature Review

Software agents are essentially applied in three types of research areas. The first type is the original applications that provide computational models of distributed intelligence to researchers. The second and third types of applications are mainly involved with the practical usages of the software agents, which are providing interactive human-computer interfaces and assisting distributed computing.

1. Distributed Artificial Intelligence

Artificial Intelligent (AI) research has been the main contributor to the field of agent research. For example, when software agents are used to simulate robots or autonomous vehicles, one type of important intelligence for the agents to have is the path planning ability. It includes the ability to determine how each agent should plan the path according to the environment and other agent’s behavior, avoid collisions and arrive at the destination. Many algorithms have been proposed for this problem. Burmeister, Haddadi, and Matylis (1997) describe a multi-agent system for implementing a future carpooling application. A traffic control method through in-
vehicle routing and navigation system was presented by Adler and Blue (2001). Arikan, Chenney, and Forsyth (2001) have presented an efficient multi-agent path planning algorithm, which creates path plans for objects that move between user defined goal points and avoids collisions.

2. Providing Interactive Human-Computer Interface

It is not uncommon that software agents interfere with human-computer interface nowadays. The annoying spywares are the examples of unfriendly interferences, in which spywares are planted to the computers to collect important personal information. Network security is a growing concern and researches have been actively carried out to address these issues (Ghosh et al., 1998; Ye et al., 2004; Borders and Prakash, 2004). On the other hand friendly agents can help with services such as information managements, particular email managers and active news readers (Maes, 1994) and active worldwide web browsers (Lieberman, 1997). Papastavrou, Samaras, and Pitoura (2000) have presented a framework with mobile agents to provide flexibility, scalability, and robustness access for distributed databases.

3. Assisting Distributed Computing

Over the past years, Brodie (1989) has frequently discussed the need for intelligent interoperability in software systems and mobile agents have been widely applied in computer network systems. As an example, Liotta, Knight, and Pavlou (1999) have reported the usage of mobile agents to improve the performance and
scalability of monitoring systems. Vilà et al (2000) have applied agents to provide dynamic virtual path management in ATM networks.

The interoperability is especially important in distributed simulation applications, in which software agents are often applied to exchange simulation data. One of the natural applications of distributed computing for speeding-up simulation studies is using distributed computing technology for the purposes of independent replications. Such an application has been discussed by Seila, Xiang and Watson (2000). They have implemented agents collecting simulation data for each simulation run and doing output analysis with the method of Independent Replications. While this is certainly an application of distributed computing technology, it should not be looked upon as a distributed simulation. An application that may be justifiably classified as a distributed simulation model has been reported by Wilson, Cybenco and Burroughs (1999). They have developed an ocean/shipping simulation case study using D’Agent of Dartmouth College, and have illustrated the benefits of using mobile agent for distributed simulation. In a subsequent paper, Wilson and her colleagues (2001) have extended the ABELS framework, which allows independently designed simulations to communicate with no a priori knowledge of the details of other simulations and data resources. We develop an architecture using data exchange agents to collect data from different data sources for statistical analysis. The analyzed data can be used by other modules in the system either to control the physical system, or to modify the decisions. In this way heterogeneous modules can be integrated within the SPEED-CS architecture seamlessly.
2.3 Summary

In this chapter we have reviewed the background knowledge on Distributed Simulation. Computer systems based on message passing and on shared memory are the two main architectures for distributed computing. Different ways to decompose a simulation have been presented. Time management has always been an important and difficult issue in distributed simulation, because each sub-simulation (or federation) has its local time. Causality errors happen when event occurrences are out of order. In order to avoid/solve causality errors, researchers have proposed many time management approaches, which can be categorized as either conservative mechanism or optimistic mechanism. In addition, we have reviewed the two types of distributed simulation paradigm: specific tool paradigm and federate paradigm. The specific tool paradigm designs a distributed simulation system with domain knowledge for certain application. Federate paradigm designs a generic distributed simulation system, in which federates and federations can join in the simulation as long as they follow a set of design specifications. The High Level Architecture (HLA) specifications, which have become an IEEE standard, are introduced. We outline the modeling and simulation formalism that is used in the SPEED-CS project: Discrete Event System Specification (DEVS), and summarize the services that DEVS provides in comparison with the HLA specifications. We also give a survey of the applications of distributed simulations in manufacturing control systems, which is one of the main application fields of the SPEED-CS architecture. Software agent research
is another important aspect in our architecture and we present background knowledge as well as a survey of its applications.
CHAPTER 3 A DISTRIBUTED COMPUTING ARCHITECTURE FOR SIMULATION AND OPTIMIZATION

In this chapter, we use the SPEED-CS framework to integrate distributed simulations and optimization models. Many problems require the integration of these two types of models. For example, stochastic programming can use simulations as a scenario generator for optimization models; in some other cases, simulations need optimization models to help configure system parameters. The SPEED-CS framework is shown to be able to provide various services to help the integration of simulation and optimization models. While we use the SPEED-CS framework as the underlying architecture for our example, it is important to recognize that the designs are generic. We illustrate our implementation with a product-mix example. The example integrates a discrete event simulation of a product-mix problem with a linear programming optimization model of such a system. The simulation updates the parameters in the optimization model, which as a result will generate a new production plan. The implementation demonstrates the ability of the architecture to integrate simulation and optimization components that are developed by different parties over homo- or heterogeneous developing environments.

3.1 Introduction

Simulation has become an important method of systems analysis, and is widely used in engineering and science. The application areas of this method include
manufacturing, financial engineering, military games, and many other fields. Meanwhile, practical optimization has seen sustained developments in all facets, including modeling, algorithms, and software. These developments together the need for advanced modeling and simulation in industry, military, and government have provided the impetus for integrating simulation and optimization for complex engineered systems.

There are several reasons motivating interactions between simulation and optimization models. Fu (2002) cites the following interactions between optimization and simulation models:

- **Simulation for Optimization.** Simulations work as a scenario generator which provides the optimization with a sample space. One of the more common settings for this arises in stochastic programming, where the term “sample-path optimization” has been coined by Robinson and co-workers. Plambeck, Fu, Robinson, and Suri (1996) apply sample-path optimization method to optimize convex stochastic performance functions. Gürkan, Özge, and Robinson (1999) broaden the method’s applicability to include the solution of stochastic variational inequalities. Such methods are also referred as “retrospective optimization”.

- **Optimization via Simulation.** In this category the optimization component orchestrates the simulation of a sequence of system configurations, so that the system configuration obtained eventually is an optimal or near optimal solution. Suri and Leung (1989) used a stochastic approximation method to optimize a
simulation model in a single simulation run of an M/M/1 queue problem. Other methods include the stochastic ruler algorithm (Yan and Mukai, 1992), variants of simulated annealing altered to accommodate randomness (Gelfand and Mitter, 1989; Gütjahr and Pflug, 1996; Arefaei and Andradóttir, 1999), and Andradóttir’s (1996) random search algorithms. Pichitlamken and Nelson (2003) report a combined procedure which consists of a global guidance system, a selection-of-the-best procedure, and local improvement for use when the performance measure is estimated via a stochastic, discrete-event simulation, and the decision variables may be subject to deterministic linear integer constraints. This particular arena of research has also attracted the attention of simulation software vendors (AutoStat, OptQuest, OPTIMIZ, SimRunner, and WITNESS Optimizer) who already provide some optimization capability, although optimality in these systems is difficult to verify.

Another setting in which it becomes important to provide services for data exchange between optimization and simulation is to enable models that accommodate multiple fidelities. For instance, tractability of an optimization model might require a coarse-grain stochastic model, whereas, a detailed (fine-grain) simulation model may be used to examine the consequences of the solution provided by the optimization model. Such an approach was reported in Sen, Doverspike and Cosares (1994), and, with increasing popularity of “fluid approximations” of queueing systems, this approach is becoming increasingly popular. The fluid approximations relax discrete nature of the objects in a flow control system and may lead to efficient approximation
or heuristics procedures and sometimes even to efficient optimization algorithms. Applications of “fluid approximations” can be found in papers reported by Chen and Mandelbaum (1991), Bertsimas and Gammarnik (1999), Boudoukh, Penn, and Weiss (2001), and others. Finer gain simulation models that capture detailed discrete object flow features are often used to evaluate the efficiency of the method.

In addition to the integration of simulation and optimization, there are also needs to integrate simulation models or optimization modules together. In large-scale system simulations, simulation models can be partitioned and developed by different groups. Simulations developed by each group are required to be able to integrate with those developed by other groups so that the entire system can be studied as a whole. Similarly in large-scale optimization, problems are often decomposed so that smaller (and perhaps easier sub-problems) can be solved. While there are many examples of efforts to solve optimization problems with distributed computing techniques, the software developed for these examples are rarely designed for re-use by different classes of models. The researches usually focus on one type of problems rather than constructing a generic architecture. Interesting research results include Lee’s (2004) implementation of a disjunctive cutting-plane algorithm in a distributed memory environment, Blomvall’s (2003) parallel algorithm for multistage stochastic programming, and so on. While it is clear that distributed computing is playing an increasingly important role for simulation and optimization, software engineering practices remain limited to customized implementation for specific applications. The efforts to develop a distributed computing system for a specific problem are usually
difficult and time consuming; therefore it is necessary to develop a systematic architecture for this purpose. The architecture is required to be composable such that different models, including decision models and simulation models can be integrated. It should be interoperable to support communication and synchronization at runtime. In other words, the architecture should be able to provide such services as to dispatch tasks, control runs, and exchange outputs of each type of model.

In this chapter we discuss the design of a distributed computing system within the SPEED-CS architecture which sets up a flexible platform that provides services to integrate simulation and optimization models. We illustrate its uses with a product-mix problem as an example. The organization of the rest of this chapter is as follows.

In the next section, we review the baseline technologies and tools that are available for the generic architecture as cornerstones. Section 3 addresses the requirements and design of the generic architecture. In section 4 we present the implementation of an experimental system with the SPEED-CS framework. We present an example of interoperability between product-mix optimization and simulation in section 5. Some concluding remarks and future directions are discussed in section 6.

3.2 Baseline Technologies: An Overview

In order to provide an environment for interoperable optimization and simulation as a models we adopt a “distributed computing” framework in which each component is treated independently, whereas they will communicate with each other through formal procedures. Whether these components actually operate on separate
processors is not very critical to our design. In this sense, the distributed computing framework is used as a methodology for structured exchanges between the alternative components. Accordingly, we begin by providing some background on distributed computing, and related baseline technologies.

3.2.1 Distributed Computing Environments

There are two principal paradigms for constructing parallel and distributed simulations: specific tool paradigm and federate paradigm. We argue that the federate approach is more appropriate for the task at hand. In order to appreciate this choice, note that the federate approach is better suited for situations in which algorithmic modules operate somewhat independently, and are able to exchange data as necessary. For this reason, Ziegler, Praehofer, and Kim (2000) also adopt this approach when integrating discrete and continuous simulations. One of the other attractive features of their work is the use of DEVS which is a system-theoretic formalism that treats each model as a composition of dynamic subsystems, each with a time base, states, inputs, outputs, and transition functions for determining next states and outputs given current states and inputs. One of the main advantages of this system-theoretic approach is that it enables model-composability through coupling atomic models, and moreover, these models are closed under coupling (i.e., the resulting model is also a DEVS model). Consequently, as long as a model can be “wrapped” using a DEVS model, the entire system can be viewed as a DEVS object. Ziegler, Praehofer, and Kim (2000) use this property to provide an interoperable framework for discrete-event, discrete-time as well as continuous-time (through
Our approach will add the dimension of decision-making to a coupled collection of DEVS models. Then, the DEVS runtime infrastructure, which provides a mechanism to federate simulators, will also be available for coupling simulation and decision models. This extension has several benefits. It can enable development of a methodology for describing models composed of decision models and process models with saving the learning curve and programming details of a given simulation engine and a given algorithm. It will make the evaluation of certain decision policies and the studies of alternative decisions for a process more convenient without requiring custom software development. Moreover decision models and simulation models that are developed by different parties can be integrated seamlessly so long as the modeling semantics are consistent. Such a collection of models can provide extensive reusability.

3.2.2 Decision Models

Decision models formalize choices with well-defined mathematical formulations. Often, these models lead to optimization problems in which a decision maker’s preferences, and resource constraints are represented mathematically. These models are then solved or approximated using the algorithmic machinery of optimization. The output of the algorithmic process is then communicated to the user through some interface which may provide capabilities to re-run the model using different parameters or scenarios. It is important to recognize that the entire modeling process involves several phases: model development, data gathering and input preparation, choice of solution methodology, and finally output analysis. As the technology for
optimization has evolved, software tools have been developed to help with the entire modeling process.

Over the past two decades, the development of modeling languages has become one of the main driving forces in the implementation of optimization models. Due to the algebraic nature of many optimization models, especially linear and mixed-integer programming models, the languages provide the user the ability to state an optimization model using algebraic constructs. Among the more widely used algebraic modeling languages there are MPL (http://www.maximal-usa.com), AMPL (http://www.ample.com), GAMS (http://www.gams.com), AIMMS (http://www.aimms.com), and so on. In the area of combinatorial or discrete optimization, where effectiveness of solution algorithms often depends on the ability of the modeler to reduce the search space (e.g. by developing strong relaxations using valid inequalities), it is useful to provide the ability to have the solution algorithm communicate with modeling language so as to allow model refinements as the algorithm proceeds through its search process (Fourer and Gay, 2002). Such procedures are available with some languages (e.g. ILOG/Concert Technologies).

While algebraic modeling languages are convenient to construct decision models, they are lack of the functionality to allow interoperability among models. There are active researches aiming to increase interoperability between different algebraic modeling languages. For example, Fourer, Lopes, and Martin (2004) have designed LPFML, an XML schema for representing linear programming instances. We develop the SPEED-CS framework to encapsulate decision models to achieve
interoperability. Our approach develops a modeling layer to provide a mechanism that allows users to develop models sharing the same communication interface. This kind of interface can readily setup communications among decision models, simulation models, as well as physical systems. Moreover the SPEED-CS framework abstracts different solvers’ functions to a series of common functions, which is indifferent of who provides the callable libraries. This abstraction is beneficiary in practice because it can achieve better speed performance than constructing text format models for the solver to load and solve the problem.

3.2.3 Network Middleware

Recent developments in computer network technology have made available numerous middleware packages that help implement distributed systems such as web services, distributed computing, multi-source data fusion, and so on. In all these applications, a middleware package helps realize both networking and communication services by taking care of details such as data conversion due to different operating systems, establishment of communication through protocol stacks, and even the transmission of signals through hardware in complex real-time systems. Although the use of middleware introduces some computation and communication overhead, it enables and simplifies how the distributed components are developed and connected. A package of middleware is required to have the following three basic elements (Buss and Jackson, 1998):
• An object interface language. An object interface language is important for supporting distributed applications with an abstractive level of communications. The object interface language defines the minimal information about the implementation of methods and members of another object on another computer.

• An object manager. The object manager is responsible for passing object references to requesting clients, instantiating objects as necessary and marshalling object requests between different machines.

• A naming service. The naming service is the mechanism by which a server informs clients about objects available for access.

Applications may be developed using either language-specific middleware as RMI (Remote Method Invocation), or language-neutral middleware such as CORBA, and DCOM. Because RMI is specifically designed for the interoperability between objects developed in JAVA language, it uses JAVA’s own interface syntax as its object interface language (http://java.sun.com/docs/books/tutorial/rmi). RMI generates stubs for the remote objects, and uses the stub class of the remote object as a proxy in clients so that these clients can communicate with a particular remote object. An application can register its remote objects with RMI’s simple naming facility, the rmiregistry, or the application can pass and return remote object references as part of its normal operation.

Among language neutral middleware CORBA has become a popular choice. Since it can facilitate connections between objects developed with different
languages, it requires an interface definition language (IDL). The IDL defines an interface that a remote object possesses. Then the IDL interface is compiled into code in one of the supported languages as the basis for the implementations. CORBA uses the Object Request Broker (ORB) to manage the remote objects. The ORB locates the remote object on the network, communicates the request to the object, waits for the results, and communicates the results back to the client when they are available. CORBA provides multiple services, among which the naming service defines how CORBA objects can be accessed with friendly symbolic names. Another commonly used language neutral middleware is DCOM (Distributed Component Object Model), which is the distributed extension to COM (Component Object model) supported by Microsoft (http://msdn.microsoft.com). DCOM builds an object remote procedure call (ORPC) layer on top of DCE RPC to support remote objects. It uses an IDL similar to the one for CORBA to define the interfaces of the remote objects.

3.3 SPEED-CS System Design

In order to achieve composability and interoperability between model components, there are four aspects of design that bear reflection. These are the design of system architecture, network programming, system modeling, and data models. System architecture designs the coordination of the components. Network programming involves the choice of communication middleware and a plan of how to partition and distribute components. System modeling specifies the mechanism of
modeling objects and coupling model components. Data models are necessary to provide a common data format for components to interact with each other. Although the actual implementation varies, any distributed system with the scope we have identified requires a resolution of these issues. In the following we first present the system architecture and then discuss the rationale for our choices.

3.3.1 SPEED-CS Architecture

The SPEED-CS system is constructed with multiple layers as illustrated in Figure 3.1. The layers include an Object Request Broker (ORB) layer, a SPEED-CS layer, a components layer, and a User Interface/Modeling layer. The coupled model is defined in the Modeling layer, while the model components are in the Components layer. The dashed line indicates that the model components are linked remotely. With the services in the SPEED-CS layer and ORB layer, a remote object can be referenced. Each layer has functionality as follows.

- **Object Request Broker (ORB) layer.** The ORB layer provides the distributed communication services. It automates many common network programming tasks such as object registration, location, and activation; request de-multiplexing; framing and error-handling; parameter marshalling and de-marshalling; and operation dispatching. The ORB layer is important in that it hides network programming from the developers so as to save time of designing, implementing and debugging the software components involved in networking.
• SPEED-CS layer. The main service provided by this layer is to manage the model components. It maintains a registry of all the model components by names, server machine URLs and ports. When there is a request for a component, the SPEED-CS layer locates the component by facilitating the ORB layer Naming Service with the information in the registry. The Naming Service associates abstract names with CORBA objects so that a client program is able to find those objects by looking up the corresponding names. A server that holds an object reference can register the object with the Naming Service, giving it a name that can be resolved later as a key to the object.

• Components layer. It maintains a collection of simulation model components and decision model components. By specifying the names and coupling relations of the components, users can construct a composite model. This is accomplished within the SPEED-CS layer which resolves the names, finds the objects, and instantiates the couplings of the components. Thus from the point of view of a user, it does not make much difference from using the local components. (Other functionalities include using CORBA services to invoke the methods of the objects and dispatching the results to other models according to the coupling information defined by users.)
• User Interface/Modeling layer. This layer provides a modeling interface for user to construct the models. High level modeling languages and graphical user interfaces (GUI) can be developed for a more user-friendly system.

There are significant benefits of designing the software system using layered architecture. One of the main advantages is that it maintains modularity, and allows operability between layers without complete knowledge of the intricacies of the other layers. This feature greatly eases modifications, updates, or even substitutions of any layer with minor or even no changes to other layers. Also the development of layers can be accomplished in parallel as long as the interfaces between layers are agreed upon.
3.3.2 Network Programming

The choice of the middleware needs to consider the programming of the modeling tools, convenience of integrating objects, and the availability of middleware. For our system, we choose CORBA as the underlying middleware since we propose to provide a framework that can integrate federations or simulators developed with C++ and Java.

Middleware offers communication services for model components within the framework. Each model component is a piece of software that performs certain functions and is encapsulated with a common interface. The interface, referred to as “coupling,” provides channels to accomplish interconnections to the other model components. The coupling generally represents data flows between components. The interconnections can be between two models on the same computer, or between two components on different computers connected via a network. The components fall into two categories: simulation model components and decision model components. Simulation components are systems that enable discrete-event simulations whereas decision model components are objects that set up the mathematical programming with input data to run algorithms which output decisions. The common interface of the simulation model components or the decision model components allows interoperability between these models. Each model component maintains a part of the state-space of the system. This method to decompose a large-scale simulation is referred as the state-space domain decomposition (Vee and Hsu, 1999). In this decomposition approach, the simulation model is decomposed into a
number of sub-models or components in space domain. Each component is assigned to one process, where several processes may be run on the same computer. The components exchange information between each other based on the coupling relations. The event list is also decomposed into individual ones to ensure thorough parallelism. This kind of decomposition enables parallelism, and is easy to implement with a federated collection of models.

3.3.3 Modeling and Simulation Formalism

In order to facilitate interoperability, we will choose a common formalism for both simulation and decision models. In the SPEED-CS system, we adopt the DEVS formalism which provides a means of specifying a mathematical object representing a discrete-event dynamic system. Such a system has a time base, inputs, states, and outputs, and functions for determining next states and outputs given current states and inputs. While decision models, especially static decision models, may not seem like discrete-event dynamic systems, it is important to view these models as components within a simulation in which they provide outputs when required. Thus, decision models will also be treated as DEVS models.

At the lowest level, an atomic DEVS model describes the autonomous behavior of a discrete-event system as a sequence of transitions responding to external input (event) and internal input (event). At the higher level, a coupled DEVS describes a system as a network of atomic or coupled models. The connections in the network denote how models influence each other (Zeigler and Sarjoughian, 2003a).
It is convenient to realize a distributed computing simulation with the Parallel DEVS simulation protocol. The Parallel DEVS protocol seeks to exploit parallelism in the simultaneous occurrence of internal and external events among many components. The parallelism is realized by a coordinator that synchronizes the components as each simulation model cycles through its steps. To start a cycle, the coordinator collects the times of next event from the component simulators. It sends the minimum of these times back to the components, thereby allowing them to determine whether they are imminent, and if so to generate output. More than one component may be imminent and the outputs of all such imminent components are sorted and distributed to others according to the coupling specification of the coupled model. The transition functions of the imminent components, as well as all other recipients of inputs, are then applied. The resulting changes in states may cause new values for time advances and these are sent to the coordinator. Processing then continues to the next cycle (Zeigler et al., 1999).

In our design, both simulation model components and decision model components are developed as atomic or coupled DEVS models. Their common interfaces allow the couplings among the model components in a distributed environment. When the coupled system runs, the components communicate with one another following the parallel DEVS simulation protocol.
3.3.4 A Common Data Model

In the process of integrating of heterogeneous applications, it is important to have a common Data Model as the understandable data format among the applications. A widely used cross-platform, extensible standard for common Data Model is the XML (eXtensible Markup Language). In the system, we use XML as the standard format to encode the data transferred between different models.

XML uses tags to mark up the documents and structure of the documents. The tags are domain-specific vocabularies, and thus this protocol can be used for different kind of applications, as long as the components in the applications can decode the meaning of the tags. The XML data forms a tree structure and is easy for the parser to search for the needed information. Each model can play the role of part of the pipeline of the XML data flow, which can extract the data elements it needs, and then filter/modify the data before passing the dataset downstream to next model. The packages of APIs to program with the XML technology provided by Java are accessible for developers.

In the SPEED-CS system, XML data streams are used to encode information. The information can be outputs generated by the simulation, or it can be user input data to formulate the models. In order to obtain the data expressed with XML format, a specific parser is designed for this task. The parser seeks keywords for extracting data. As an example, a linear programming model encoded as a XML data stream should express information such as whether it is a maximizing or minimizing
problem, how many variables the model has, and others. Therefore the keywords of “maxOrMin”, “numberOfVars”, and others are introduced to the parser (see Figure 3.3). With these vocabularies, the parser is able to extract information and thus to construct the LP model for the solver.

3.4 Implementation

The implementation of the architecture involves in choosing the middleware, developing the component management layer, component layer, and the modeling layer. We begin the illustration of the implementation with the ORB layer, which provide middleware services for the system. Then we present how we model the decision model components followed by the introduction of the SPEED-CS layer that works as the model manager. Before we finally show the integrated system, we discuss the construction of simulation models.

3.4.1 Object Request Broker (ORB) layer

The main service that the ORB layer provides is the object broker services. While it is not impossible to develop the ORB layer from scratch, it is more convenient to inherit from the commercial or off-the-shelf middleware. Since the CORBA technology is mature and widely available, we adopt the CORBA technology as the middleware for the system. One of the major advantages is that the system developed on top of the CORBA middleware can be heterogeneous. More specifically, the system is able to integrate simulation model components and decision model
components developed with different programming languages. The decision model can be developed with either user-developed algorithms, or with some callable libraries that are provided by vendors. The simulation model components may be developed by various methods and tools but need to follow the DEVS formalism. In our examples DEVSJAVA is the modeling tool for the simulation models implemented in JAVA language. JAVA is an object-oriented programming language that blends well with CORBA by complementing each other. We choose to use JAVA to develop the SPEED-CS layer to take advantage of the distributed interoperable framework and services that CORBA provides as well as inherit the portability and robustness of JAVA.

There are multiple options of CORBA middleware to choose from, among which OmniCORBA is used in our implementation. OmniCORBA implements the 2.3 specification of the Common Object Request Broker Architecture (CORBA), that is, the Internet Inter-ORB Protocol (IIOP). The protocol provides omniORB the means of achieving interoperability with the ORBs implemented by other vendors. Moreover, the IDL to C++ language mapping provided by ominORB conforms to the latest revision of the CORBA specification. The omniORB is multithreaded and portable on UNIX, Windows NT and 95. The omniORB provides an implementation of OMG’s COS Naming Service Specification. This feature allows the client program to use a human readable name and the conversion into an object reference is hidden underneath the normal invocations (http://www.uk.research.att.com/omniORB).
3.4.2 Decision Model Components

Decision Model Components are objects that provide decisions, usually solutions of an optimization model, to the system. A decision model component contains APIs to input model data, run the algorithm, and output solutions. In order to deploy over the network, a decision model component needs to export a few functions for remote invocations. In the following we present the design and implementation of using linear programming as a decision-making paradigm. Other decision models can be developed in a similar manner. An LP model contains a set of decision variables, an objective function and a set of constraints. The following formulation represents a general LP.

\[
\text{objective function : } \min \ c^T x \\
\text{subject to : } A_1 x = b_1 \\
A_2 x \geq b_2 \\
A_3 x \leq b_3 \\
lb \leq x \leq ub
\]

where \( x \) is the decision variables, and vectors \( c, lb, ub \), and matrices \( A_1, A_2, A_3 \) are coefficients.
Equations and inequalities form the set of constraints. Instead of writing a model file in certain common format (such as MPS), the LP model component provides a generic skeleton of a linear programming for callable functions of solvers. Modelers formulate the linear programming and prepare the input dataset. The process of running a linear programming model can be abstracted as the following sequential process.

```plaintext
module genericLP
{
    typedef sequence <double> arr;
    interface genericLPprob
    {
        short initialization(in short n,
                             in short m1,
                             in short m2,
                             in short m3,
                             in boolean maxOrmin);
        short loadObj(in arr costs);  //size is defined by n
        short loadAXleB(in arr A, in arr B); //AX <= B
        short loadAXgeB(in arr A, in arr B); //AX >= B
        short loadAXeqB(in arr A, in arr B); //AX = B
        short loadub(in arr ub);  //X's upper bound X <= ub
        short loadlb(in arr lb);  //X's lower bound X >= lb
        short setInt(in arr intflag);  //sets integer variables
                                        //solution
        short optimize(out string name, out arr act, out arr duel);
    }
};
```

Figure 3.2 A linear programming interface

- Initialization. This procedure sets the solver environment and gives necessary information to the solver such as the number of decision variables, the numbers of equations and inequalities, and the sense of the objective function (i.e., maximize or minimize) the problem. Solvers may need these data to estimate the size of the
problem. They reserve memories based on the problem size and the particular algorithm of the solver.

- Data Input. This procedure populates the component with data to construct decision model, including the objective function, constraints and bounds. For integer or mixed integer programming, the input data needs to indicate to the solver which decision variables should be treated as integer.

- Solve. This procedure solves the linear programming problem after the model is set up. User may be able to choose different algorithms to use. For example, many linear programming solvers allow the user to choose either Simplex method or Interior Point method. It is the user’s decision to apply which algorithm to use in solving the problem.

- Output and Sensitivity Analysis. The component outputs the activities and duals when the solve process is accomplished. It can also output the status of the solver and model.

Figure 3.2 is the interface definition of a linear programming model component, with the standard interface definition language (idl). The procedures are specified with several methods. When compiling the interface file to a programming language, a server program and a client program will be generated. The server program implements the methods and the client program packages the component to a DEVS atomic model. The atomic model can be coupled with other DEVS models to construct more complicated models. When the atomic model (client) receives an
event from another DEVS models, it will respond with the state transitions. The change of the model state may result the running of one of the functions defined in the interface file. For example, the client can be in one of the following states: “passive”, “initializing”, “waiting for data”, “loading data”, “optimizing”, and “done”. Initially the state of the component is “passive”. Driven by the event from another model to initialize the algorithm, the component transitions to “initializing” and thus invokes the initialization function. After the initialization is done, the internal state transition updates the state to “waiting for data”, which means the client is ready to accept input model data. The transitions go on until the model is solved and the client remains “passive” state unless another event activates the component.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<initialization>
  <numberOfVars>10</numberOfVars>
  <numberOfLE>0</numberOfLE>
  <numberOfGE>0</numberOfGE>
  <numberOfEQ>3</numberOfEQ>
  <maxORmin>true</maxORmin>
</initialization>
```

Figure 3.3 An example of data stream to initialize an LP

The data received from other components are encoded as a DEVSJAVA message in XML format. Figure 3.3 shows an example of the data stream for the initialization process. The initializing data stream specifies the number of variables, the number of “less than” constraints, the number of “greater than” constraints, the number of equations, and the sense of the problem. Although different solvers have different needs for memories, problem size which is indicated by these numbers is one of the
main indices for the solver to estimate the memory requirement. The LP component parses the data and if the data are valid it will call the initialization method defined in the interface file to accomplish the initialization process. If the method is called successfully, the LP component will then transition to the state “waiting for data”. When the expected data are injected as another event, the LP component will populate the initialized model with objective coefficients data, the constraints data, and the bounds data, and so on. In our example, we have 10 coefficients for the objective function. The objective XML data stream is illustrated in Figure 3.4. Note that a decision variable name is an attribute of a reserved tag “LPVAR” and the content of the element is the coefficient of the decision variable. In a similar way the data for the constraints and bounds can be input to the model.

```
<Objective>
  <LPVAR name="x1">3.0</LPVAR>
  <LPVAR name="x2">3.0</LPVAR>
  <LPVAR name="x3">3.0</LPVAR>
  <LPVAR name="x4">3.0</LPVAR>
  <LPVAR name="x5">2.0</LPVAR>
  <LPVAR name="x6">3.0</LPVAR>
  <LPVAR name="x7">2.0</LPVAR>
  <LPVAR name="x8">-4.0</LPVAR>
  <LPVAR name="x9">-4.0</LPVAR>
  <LPVAR name="x10">-3.0</LPVAR>
</Objective>
```

Figure 3.4 An example of objective function data stream

Using standard XML toolkits, we are able to validate and parse the data from the inputs, and then store them in proper arrays. After the model is constructed, the component will receive the “optimization” event to run the model. The component
thus transitions to the state of “optimizing”. When the solving process is done, the solution, including activities and dual values are obtained. The solutions are converted to the XML format and output to designated components.

Other decision models can be developed as a component by defining event sequences for the procedures of handling data and running models. A decision model skeleton is defined to export all the necessary functions to set up the model. The user populates data to the model and invokes the event sequences though the DEVS client of the component, whereas the client can be coupled with other DEVS models. This approach can lead to a growing component set with more decision model components joining in. The set can become a distributed shared resource.

3.4.3 SPEED-CS Layer

The SPEED-CS layer is the software to manage the components. The management includes maintaining a list of components, adding or deleting components, and initiating access of remote objects. There are two equally important services supporting the management: the naming service and event service. The naming service provides users with a way to develop applications within a distributed computing environment, without sacrificing the advantages of a local environment. The event service provides controls over the execution of the whole system.

1. Naming Service

CORBA offers the basic naming services and the SPEED-CS layer extends the CORBA naming services to manage the components by maintaining a list of the
properties of the components that users may want to reference and invoke. The property list includes the name of the component, the name or IP address of the server computer on which the component is implemented, and the port of the server.

We design a static class SPEEDCS to check the user-input name with those in the property list. If the user-input name does not match the names in the property list, it means that the requested component is not available. The request of the component is then rejected and an exception is thrown to inform the user that there is a mismatch of the name and component. If there is a match, it means the component is available. The SPEED-CS layer will use the name as a key to query the information about the computer and the port on which the component server is located. Then the SPEEDCS class creates an instance of GenericClient with the component name, the port, and computer name in the property list. The client instance will connect the server on the computer when the component is requested. For example to create an LP model component prodmixprob that utilizes the linear programming model, we use the following code:

```java
    GenericClient prodmixprob = new GenericClient(2809,
            "hawk",
            "genericLP.genericLPprob");
```

which indicates that the object can be referenced at the port 2809 on the computer named “hawk”. It is a CORBA object with name of the component as genericLP.genericLPprob.
The SPEED-CS layer maintains the client programs generated from the idl (interface definition language) scripts of the decision model components. The client programs include the standard java code generated by the idltojava compiler such as stubs, skeletons and object helper. In addition to those objects, another object named “SpeedCSHelper” is also generated, whose main functionality is to convert the CORBA object to a JAVA object. As shown in Figure 3.5, after the user passes in the name of a component in the nc_array, GenericClient first resolves the names and obtains a CORBA object initRef. It then instantiates the SPEED-CS helper class and invoke its method "narrow" to obtain the JAVA object.

```java
org.omg.CORBA.Object initRef =
    namingContext.resolve(nc_array);
    
Class helper = Class.forName(objName+"SpeedCSHelper");
objHelper = helper.newInstance();
Class[] args= {org.omg.CORBA.Object.class};
java.lang.reflect.Method _narrow =
    helper.getMethod("narrow",args);
obj = _narrow.invoke(
    objHelper,
    new org.omg.CORBA.Object[] {initRef});
```

Figure 3.5 Locating the generic LP object

The above operations are totally packaged within the SPEED-CS layer and the user does not experience much difference from a normal local programming. An example of user request a remote LP decision model looks as follows:
genericLP.genericLPprob lpProblem = SPEEDCS.instanceOf("genericLP.genericLPprob");

Where `instanceOf` is a static method of the `SPEEDCS` class to accomplish the operations connecting, binding, and reference by name. After the remote object is defined, the invocation of the methods is just the same as for a local object.

It is a straightforward process to maintain the property list. If new components are added to the server, the administrator just needs to add to the property list the name, server machine on which the object is located, and the ports to connect. In this way the SPEED-CS layer is open to variant CORBA objects of different decision models and simulation models.

2. Event Service

The event service in the SPEED-CS platform is realized by the DEVS formalism. DEVS provides an event-based state system, in which the states of an object transition according to the events scheduled as responses to received messages.

Time-management is an important issue to implement with event services. One of the ways is to maintain the SPEED-CS layer clock as a universal clock and thus it can monitor the time advance and event scheduling. Both the decision and simulation model components are developed as DEVS models. The model therefore enables the component to utilize DEVS functionalities, such as state transitions, receiving and sending messages, and reacting to received messages, etc. If a received message is to call a method of the remote component, the component first references the remote model by utilizing the naming service and then invokes the method. Modeling with
the DEVS formalism also facilitates the hierarchical modeling to couple the components to larger coupled models.

3.4.4 Simulation Components

Simulation model components are another important category of the components in the architecture as much as decision model components. The design of simulation components is very similar to that of the decision model components. A simulation component also includes a server program and a client program. The server is modeled with DEVSJAVA and implements the discrete-event simulation application, and the client provides the naming service for users to locate the component. The idl interface declares the DEVS simulator methods, such as executing transition functions, getting output from the models and so on. The SPEED-CS coordinator will call these methods to provide event services when running the integrated system.

The simulation models share the DEVS hierarchical modeling scheme. A DEVS model specifies 1) basic models from which larger ones (coupled models) are built, and 2) how these models are connected together in hierarchical fashion. Larger models can be constructed by coupling atomic models or smaller coupled models.
3.4.5 Integrated System

The entire integrated system is put together as shown in Figure 3.6 and Figure 3.7. Figure 3.6 shows the process that a simulation model invokes the decision model. The user specifies the relation between the decision model components and simulation components by providing the system with the coupling information. As indicated in the Components layer, the dotted line means the connection is a remote one. When a simulation model needs to pass data to a decision model, it first resolves the name of the decision model utilizing the naming service in the SPEED-CS layer. After the remote decision model is located, the data are transfer through the ORB layer to invoke the methods of the decision model.

![Diagram of the integrated system](image)

Figure 3.6 A simulation model invokes a decision model
Figure 3.7 illustrates the primitive procedures when the decision model outputs the decisions to invoke the simulation. Using naming services provided by the SPEED-CS layer, the remote simulation model component is located. The decisions are passed to the simulation component as parameters for the simulation to run.

Because the output data from the decision model component cannot be used directly by the simulation models usually, it is necessary to convert the data to the context of the simulation data. The same situation exits where the simulation output data need to be converted to the semantics that the decision model can understand. Therefore we introduced the converters: “LToSim” and “SimToLP”. More details about these two modules will be described in the next section when we present an
example, yet it is important to point out that the converters are designed and implemented for general simulation and decision model components.

Interconnections of decision models only or simulation models only can be realized with the framework readily. Couplings of decision models only can be used in cases where a large-scale optimization problem with a structure appropriate for decomposition needs to be solved using a decomposition technique. Similarly, couplings among simulation models lead to a distributed simulation. The flexibility and convenience enables the SPEED-CS architecture to handle applications requiring coordination between various decomposed models. Another benefit of this architecture is that the set of components can be enlarged by adding decision model components and simulation model components. By using a collection of standard components, software implementation of models will be no more than a task of defining the relationships between the components.

3.5 An Example

In this section, we illustrate the system by presenting an example of product-mix optimization and simulation as an interacting system. A product-mix problem (Schrage, 1997) has a collection of products that compete for a finite set of resources. Associated with each product is a profit contribution per unit, and associated with each resource is availability. The objective is to find how much to produce of each product so as to maximize profits subject to resource constraints. A discrete event simulation will be created for the product-mix problem, where processing time of
each product on each machine is a random variable whose distribution is known. However, because of queuing delays, the distribution of the total time at a machine is not known a priori. Because of this, the product-mix model uses an estimated average processing time for each machine, and this estimate is updated through interactions between the LP formulation and the discrete-event simulation.

3.5.1 The LP formulation

In this particular example there are 3 machines and each machine can produce 7 types of products. The machines are not identical and they process each type of product with different processing times. We assume that the jobs are processed with the sequence of operations on machines 1, 2, and 3. Initially the decision model has an estimation of the expected processing time as shown in Table 3.1. The simulation model has another table of expected processing time as shown in Table 3.2. The net profit of each product is shown in Table 3.3. Both the decision model and the simulation model share the same data for profit.

<table>
<thead>
<tr>
<th>Products</th>
<th>Machines</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>5.0</td>
</tr>
<tr>
<td>B</td>
<td>7.0</td>
</tr>
<tr>
<td>C</td>
<td>8.0</td>
</tr>
<tr>
<td>D</td>
<td>7.0</td>
</tr>
<tr>
<td>E</td>
<td>6.0</td>
</tr>
<tr>
<td>F</td>
<td>7.0</td>
</tr>
<tr>
<td>G</td>
<td>7.0</td>
</tr>
</tbody>
</table>

Table 3.1 Initial expected processing time in minutes for LP
Table 3.2 Expected processing time in minutes for simulation

<table>
<thead>
<tr>
<th>Products</th>
<th>Machines</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td>12.0</td>
<td>8.0</td>
<td>5.0</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>7.0</td>
<td>9.0</td>
<td>10.0</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>8.0</td>
<td>4.0</td>
<td>7.0</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td>10.0</td>
<td>5.0</td>
<td>3.0</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>10.0</td>
<td>6.0</td>
<td>3.0</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td>7.0</td>
<td>11.0</td>
<td>2.0</td>
</tr>
<tr>
<td>G</td>
<td></td>
<td>7.0</td>
<td>11.0</td>
<td>2.0</td>
</tr>
</tbody>
</table>

Table 3.3 Net profit of each product

<table>
<thead>
<tr>
<th>Decision Variables</th>
<th>Definition</th>
<th>Profit per Unit</th>
<th>Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Number of units of A produced per week</td>
<td>$3.0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Number of units of B produced per week</td>
<td>$3.0</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>Number of units of C produced per week</td>
<td>$3.0</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>Number of units of D produced per week</td>
<td>$3.0</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>Number of units of E produced per week</td>
<td>$2.0</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>Number of units of F produced per week</td>
<td>$3.0</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>Number of units of G produced per week</td>
<td>$2.0</td>
<td></td>
</tr>
<tr>
<td>M₁</td>
<td>Hours of machine 1 used per week</td>
<td>-$4.0</td>
<td></td>
</tr>
<tr>
<td>M₂</td>
<td>Hours of machine 2 used per week</td>
<td>-$4.0</td>
<td></td>
</tr>
<tr>
<td>M₃</td>
<td>Hours of machine 3 used per week</td>
<td>-$3.0</td>
<td></td>
</tr>
</tbody>
</table>

Note that there are several data mismatches, and such discrepancies arise in instances where the optimization model and the simulation model are “owned” by different groups within the organization. Alternatively, one may view the simulation as representing the real world, and the LP model simply an outdated approximation.
This exercise then illustrates how the SPEED-CS architecture allows the decision model to match up with realistic data.

Some other constraints include: at most 20 units each can be produced of products D and E, and each machine can be run 128 hours (a week). There are no lower bounds for the products or the use of machine time. Let \( x_i, i = 1..7 \) be the decision variables of the number of product (A to G) to produce, and let \( x_i, i = 8..10 \) be the time used on machine 1 to 3. To simplify the problem we model the problem with linear programming and round \( x_i \) to integers. The LP model can be represented with equation constraints as follows:

Maximize \( \sum_{i=1}^{10} p_i x_i \)

subject to

\[
A_{eq} x = r_{eq}
\]

\( lb \leq x \leq ub \)

where \( x = [x_1 \ x_2 \ \cdots \ x_{10}]^T \) and \( p_i \) is the profit of each product and the cost of running each machine.

The initial equation coefficient matrix is:

\[
A_{eq} = \begin{bmatrix}
5.0 & 7.0 & 8.0 & 7.0 & 6.0 & 7.0 & 7.0 & -60.0 & 0 & 0 \\
5.0 & 7.0 & 7.0 & 5.0 & 6.0 & 5.0 & 10.0 & 0 & -60.0 & 0 \\
5.0 & 7.0 & 7.0 & 3.0 & 3.0 & 2.0 & 2.0 & 0 & 0 & -60.0
\end{bmatrix}, \quad \text{and}
\]

the equation’s right hand side is \( r_{eq} = [0 \ 0 \ 0]^T \).
The lower bound of the decision variables is specified as vector \( lb \), which is 0 for both the product variables and machine time usage variables in this example. The upper bound is specified as vector \( ub \): for product \( D \) and \( E \) (\( x_4 \) and \( x_5 \)) the upper bound is 20.0 and for machine time usages variables the upper bound is 128.0. All variables are non-negative.

3.5.2 Transferring data from the LP to Simulation

The optimization results are returned with the notations of \( x_i \), while the simulation requires some meaningful descriptions of the results. For this purpose a converter is designed to interpret the decisions to the variables which the simulations can understand. A map file helps the converter link decision variables and simulation variables. The decision variable names are defined as the attribute of the tag “LPVAR”. As shown in Figure 3.8, the simulator uses variable “JobA” for decision variable \( x_i \) to generate job type A. With the mapping the simulator is able to convert the decisions from the LP models to the simulation inputs.
3.5.3 The Simulator

We design a discrete event simulation model to simulate the running of the production system. A job generator generates each kind of jobs with a uniformly distributed inter-arrival time within the range [15.0, 25.0]. The jobs must be processed by machine 1 first, then machine 2, and finally machine 3. Each machine maintains a queue for the jobs and the jobs are lost if the queue is full. Figure 3.9 depicts the simulation setup in the product-mix example.
The machines are not identical and a machine can process all the types of jobs. A machine’s processing time of a certain type of job is a random number generated from a uniform distribution whose expected processing time is shown in Table 3.2. The production plan comes from the LP decision and the jobs are generated according to the plan. In the simulation, processing time intervals (including delays) of each job on a machine and the counts of each type of jobs are observed. The time intervals are averaged by the number of jobs produced and the results will be sent back to the optimization component for revisions of the production plan. The revised production plan will then trigger a new simulation.

Figure 3.9 Product-mix problem simulation
3.5.4 Transferring data from Simulation to LP

The LP model requires generic inputs such as equation matrix, bounds, right hand side vectors, etc. The simulation provides application-related output data, and in this example, the simulation updates the average processing time of each job on each machine.

```
<map>
  <from>Simulation</from>
  <to>LP</to>
  <EquationMatrix incomingName="avgTime"
                   columnIndex="productid"
                   rowIndex="machineid"
                   outputName="Aeq">
  </EquationMatrix>
</map>
```

Figure 3.10 The mapping from simulation to LP

A converter to translate simulation output to the LP model is therefore necessary. In the implementation, the construction of an LP is partitioned into tasks of loading objective functions, equation matrix, bounds, etc. Therefore we introduce a few keywords for data prepared for these operations. The keywords include “EquationMatrix”, “UpperBounds”, “LowerBounds”, and so on, which are related to the partitioned LP tasks. Other keywords are “columnIndex” and “rowIndex”, which is used to map the simulation data to the position of the data in matrix (or vectors) for the LP model. When the converter receives the output data from the simulation, it
extracts the data according to the relationship defined in the map file and modifies the coefficient matrices or vectors for the LP.

```
<avgTime>
  <data productid="1" machineid="1"> 11.97 </data>
  <data productid="1" machineid="2"> 23.07 </data>
  <data productid="1" machineid="3">  4.97 </data>
<!- other data points -->
  <data productid="4" machineid="1">  9.76 </data>
  <data productid="4" machineid="2">  4.82 </data>
  <data productid="4" machineid="3">  2.90 </data>
<!- other data points -->
</avgTime>
```

Figure 3.11 Average time data

Figure 3.10 shows the map file for this example. The simulation outputs the average processing time as shown in Figure 3.11, which is prepared for the equation matrix $A_{eq}$ in the LP model. Each data element is the average time for a certain job spent on a machine. The job type and machine id are specified in the data attributes. However the tag “avgTime” is not meaningful to the LP model, which is expecting “EquationMatrix” tag with “columnIndex” and “rowIndex”. With the help of the map file, the converter can modify corresponding data elements in the equation matrix $A_{eq}$. The updated matrix will be input to the LP model to modify the constraints $A_{eq}x = r_{eq}$. The updates of bounds and other data entries for the LP model are similar. The map file provides modelers with flexibility to interpret application-related simulation data to generic LP inputs.
3.5.5 Simulation Results

In the product-mix example, we design two cases in order to observe the closed-loop simulation-LP system. For both cases we keep job generation process the same and tune the variance of the machine processing time intervals. In one case, the machine processing time intervals are random numbers uniformly distributed with the expected processing time and $\pm 10.0\%$ deviation, whereas in another case the deviation is as much as $\pm 20.0\%$. In the first case the system starts with data in Table 3.1 and generates the initial production plan as shown in the first row of Table 3.4. Then the simulation runs with the plan and the processing time intervals are observed. The average processing time intervals are very close to the data shown in Table 3.2 because there are no delays and the system converges after 6 iterations. The updates of the production plan are shown in Table 3.4.

In the second case the machines process jobs with more variance while the jobs are generated with same rate as in case 1. The simulation starts with the same initial plan as in the first case. The variance contributes causes delays in the system, which as a result contributes to differences of the averaged processing time intervals from case 1. As shown in Table 3.5, the system takes more iterations to converge and it converges to a different plan from case 1 because delays affect the processing time intervals. The cases of the example have been run with different sets of seeds and the evolvements of the production plans are consistent for the runs.
Table 3.4 Production plans with ±10.0% deviation from the average processing times

Table 3.5 Production plans with ±20.0% deviation from the average processing times

The example shows the ability of the architecture to achieve interoperability between LP model components and simulation components to form more complicated
systems. If the LP model is complicated, the convergence of the system can become
difficult, and in some situations the LP may not be able to find a feasible solution. The convergence is highly related to applications.

3.5.6 Class Diagram

Figure 3.12 shows the class diagram of the example, and we only included important classes and their relationships. The purpose of a class diagram is to depict the classes within a model. Because we use DEVSJAVA as the modeling tool, many of the classes inherit from the atomic model class ("atomic") and coupled model class ("digraph"). The input and output couplings of these classes form a model component. The components provide remote methods for the discrete event simulation procedures, and the RMIClient (simulation coordinator) coordinates the execution of the distributed simulation.

3.6 Summary

This chapter proposes a distributed programming architecture for operations research studies that involve optimization models and discrete event simulations models. The architecture is a multi-layer system, which is composed of a middleware layer, the component management layer, component layer, and modeling layer from bottom up. We demonstrate the design of such a system by discussing a product mix optimization and simulation interactive system. The example includes the implementation of each layer, the methods to manage each layer, and the converters between LP and simulations. The system is able to provide services to facilitate
distributed computing, event services, naming services, and component management. We use XML as the common data format for the components. Another important feature is that the component sets can be updated and enlarged with different models adding in, as long as the models can be modeled as a discrete event model. This feature, however, raises the important issue of security of the system. There have been many research results on generic security issue on network systems, which can be implemented in the SPEED-CS architecture in the future work.
Figure 3.12 Decision and simulation integration class diagram
CHAPTER 4  A MULTI-AGENT DATA COLLECTION ARCHITECTURE FOR DISTRIBUTED SIMULATIONS

In this chapter, we present the SPEED-CS architecture as multi-agent system for data collection and output analysis within distributed simulations. Typically, a distributed model is decomposed into sub-models that exchange entities such as system states, transitions, state durations, objects etc. These sub-models may run on different computers connected through a network. Because each sub-model operates in a relatively independent manner, they each have only a partial view of system characteristics. Hence it becomes necessary to provide services that facilitate data collection and analysis within distributed simulations. One of the main contributions of the SPEED-CS architecture is to enhance the software support to enable data collection and analysis within the distributed simulation. In this chapter, we implemented a framework with the DEVSJAVA environment. The framework provides a system-theoretic basis for formulating discrete event simulation models through a collection of objects. The framework also provides a scalable distributed computing architecture. The operation of the framework is illustrated through an example in automated mining. We compare the architecture with a "baseline" methodology in which all sub-models report all data to a central database for output analysis. The experiment results show that the SPEED-CS architecture can reduce network traffic significantly, provided the data collection agent is able to choose its
operational parameters appropriately. The experiments also show that the system is able to work well with both heavy and light network traffic cases.

4.1 Introduction

Computer simulations play an important role in industrial engineering, management, military, and even video games. For example, prediction of weather conditions is made possible because of the use of simulations that approximate continuous endogenous variables representing the evolution of temperature, humidity and pressure. Such a simulation model could run on one computer if the model is relatively simple. However, to make more accurate predictions, more detailed and real world characteristics must be included in the model. Simulation models that grow in this way can easily exceed the computational capability of one computer. In other applications, the structure of the real system is based on a distributed architecture. In both situations, distributed simulations provide the appropriate modeling technology. Distributed simulation refers to technologies that enable a simulation program to execute on a computing system containing multiple processors, such as personal computers, interconnected by a communication network (Fujimoto, 1999).

Distributed simulation has been applied in a variety of areas including traffic control, battlefield simulations, network communications and so on. Consider the scenario involving traffic control in a metropolitan area. Because detailed simulation of a large metropolitan area may lead to a very large simulation model, it may be
divided into several sub-areas. Traffic control is often performed in a distributed manner; therefore a distributed simulation model offers a natural paradigm for this application. Moreover in cases where the complete simulation requires very significant computation, distributing the simulation processes reduces the execution time, yielding more frequent data that can be used for decision-making in real-time.

In some complex systems, a project may need to integrate technologies from different manufacturers separated by large geographical distances. For instance, in a warship, the radar system and tracking technology may reside on the east coast of the US, while the interceptor missile technology may be owned by a separate organization on the west coast of the US. These two technologies can be integrated in a distributed simulation environment, where each organization retains control of their proprietary information embodied in the distributed components of the model. The distributed models exchange information entities over a communication network as the simulation proceeds, allowing the troubleshooting and development of algorithms to continue without integration of the physical technologies. There are many more scenarios requiring a distributed simulation setup, such as supply chain management systems, traffic control systems, web-based education systems, etc. There are at least four principal benefits for distributed simulations: 1) reducing execution time; 2) advantageous for geographical distribution; 3) integrating simulators that execute on machines from different manufactures; 4) enforcing fault tolerance (Fujimoto, 1999).

In simulations, obtaining and analyzing data is a significant issue. These tasks are more complicated in distributed simulations than in centralized simulation, because
the data is also distributed. In this regard, some of the questions that arise include: How do we acquire relevant data? To what degree should we analyze the data in a distributed manner? What features should the software include so as to handle the distributed data collection and how to model this software? How does the data transmission affect the network traffic? In this chapter we address these questions and propose an agent-based architecture for data acquisition and analysis.

The remaining sections of this chapter are organized as follows: In the next section, we discuss the application of software agents in distributed simulations and present the idea of a data exchange agent. We model the data exchange agent as an automaton and implement it in DEVSJAVA (Zeigler et al., 2000). Then the data exchange agent is used to collect information of relevance for our analysis. Section 3 presents the setting within which the methodology is tested. We also describe our experimental approach, and discuss data needs for the experiment. Our observations regarding the experiment and our analysis are also presented in this section. The final section concludes the chapter and presents the directions for the future work.

4.2 A MAS for Distributed Data Collection

4.2.1 Design Issues for Data Exchange Agent

Several design issues need to be considered when designing the data exchange agent. Some of these issues are common to the design of agents in general (Nwana and Ndumu, 1999).
• The tasks to be accomplished. The data exchange agent, as with other agents, needs some means of discovering what the relevant resources are, and where they can be found. In addition, the data exchange agent needs to have the ability to connect to a database and search for relevant data pieces according to certain criteria.

• The realization of mobility. Because the agent needs to interact with processes on different computers, communication has obvious importance. The choice of the communication middleware depends on protocols, methods to reference remote objects, developing languages and platform, etc. For example, CORBA uses standard Internet Inter-Orb Protocol (IIOP) and a CORBA client interacts with a remote object by reference. Java RMI (Remote Method Invocation) currently uses the Java Remote Messaging Protocol (JRMP), though Sun and IBM have announced plans to enable RMI to use IIOP protocol for communication with CORBA-compliant remote objects in the future. RMI enables a client to interact with a remote object by reference, or to download it and manipulate it in the local runtime environment by value (http://java.sun.com). Considering the differences and especially the fact that most objects in our architecture are coded in Java, RMI is used as the communication interface. On the other hand, if the data exchange agent is required to be a CORBA object, it can also be implemented.

• The information format. It is necessary to have a protocol of information format to use in the distributed processes and agents. There should be a common format for agents to search and parse the data entries achieved from distributed processes.
In our current implementation, we simply use tokenized strings as the common format that the data exchange agent is able to parse. For a more general application, XML offers a good solution to encapsulate information. XML is widely applied in many data-related operations. Java has APIs to help visit the DOM trees, obtain attributes, and parse information; hence it fits well with distributed simulation architecture.

- **The intelligence level.** The data exchange agent should have certain level of intelligence to make decision if the data are sufficient for an abstraction or it needs to move on with the data. Higher intelligence level for the data exchange agent may include the ability of finding the paths by its own judgments, adjusting parameters according to the simulation data volume, and so on. We implement the data exchange agent with basic intelligence level, which serves in the experiments to prove the concept.

- **Other issues.** In order to design architecture with multi-agent data collection system, it is required to consider issues such as the traveling pattern, the interface with user queries, and so on. There should be convenient ways for the users to specify the data exchange agents’ visiting path as well as define the relevant data for the agents to collect.

### 4.2.2 A Multi-Agent System

The architecture of the multi-agent system is shown in Figure 4.1. In the design, different simulation models are distributed in the network system. The simulations
interact with each other through the transfer of objects and the exchange information between the models. The communications among simulation processes are realized by the computer networks and middleware. In each simulation, there is a *StateRecorder* that helps the operation on the data. The *StateRecorder* is a static agent with the ability to record the state change of models and to extract the data when needed. Just as the data exchange agent, it is not actively involved in the simulation applications. Whenever a model changes its current state, it reports the new state and the simulation clock at that moment to the *StateRecorder*. Upon receiving the data, the *StateRecorder* generates a SQL insertion script to put the data in the state database. The *StateRecorder* is also able to communicate with the incoming data exchange agent. It receives the queries of the data exchange agent and then converts them to SQL selection scripts. The *StateRecorder* reports the results of executing the queries to the data exchange agent.
As shown in Figure 4.1, the simulation that contains the UserInterface and OutputAnalyst is called “base” simulation only because the agent is released from and returns to this model. The UserInterface reports the user’s requests to the agent and releases the agent for information retrieval. After the agent visits all the simulation databases, it comes back to the base simulation and uploads the acquired data to the OutputAnalyst for analysis. The data exchange agent travels along the dotted arcs shown in Figure 4.1 to visit the simulation models according to a certain pre-defined sequence.

Figure 4.1 Multi-agent system
The architecture is a multi-agent system (MAS) (Jennings et al., 1998). A MAS can be defined as a loosely coupled network of problem solvers that work together to solve problems that are beyond the individual capabilities or knowledge of each problem solver (Durfee and Lesser, 1989). The designed architecture possesses the characteristics of MAS (Jennings et al., 1998):

- It contains both mobile and static agents; each agent has some tasks to accomplish. Each StateRecorder is associated with sub-simulation model, which observes incomplete information of the sub-simulation. The data exchange agent combines the localized distributed information into useful data.
- The computation of the sub-simulation model is asynchronous, that is, each sub-simulation can run with its own simulated clock.
- It is obvious that data is decentralized. Each sub-simulation contains a local state database.
- There is no global system control. Sub-simulations run on its own scheduled events and respond to the external events.

4.2.3 The Data Exchange Agent Automaton

The Data Exchange Agent is modeled as an I/O automaton developed in Lynch (2000). This approach provides a formal model associated with distributed systems. An I/O automaton models a distributed system component that can interact with other system components. It is a simple type of state machine in which the transitions are associated with named actions. The actions are classified as either input, output, or
internal. The inputs and outputs are used for communication with the automaton’s environment, while the internal actions are visible only to the automaton itself (Lynch, 2000). The automaton is responsible to the inputs by specifying the internal and output actions. An I/O automaton $A$ consists of five components (Lynch, 2000):

- A signature: $S = \text{sig}(A)$. The signature is a triple consisting of three disjoint sets of actions: the input actions $\text{in}(S)$, the output actions $\text{out}(S)$, and the internal actions, $\text{int}(S)$.
- A set of states: $\text{states}(A)$.
- A nonempty subset of $\text{states}(A)$ known as the start states or initial states: $\text{start}(A)$.
- A state-transition relation: $\text{trans}(A)$, which have the property that for every state $s$ and every input action $\pi$, there is transition $(s, \pi, s') \in \text{trans}(A)$, where $s'$ is the state after the transition and $s' \in \text{state}(A)$.
- A task partition: $\text{tasks}(A)$, which describe the thread of control over the automaton.

By adopting this formalism, we are able to ensure properties such as fairness and liveness. Fairness basically means that each task of the automaton gets infinitely many opportunities to perform one of its actions. Formally, an execution fragment $\alpha$ of an I/O automaton $A$ is said to be fair if the following conditions hold for each class $C$ of $\text{tasks}(A)$:

1. If $\alpha$ is finite, then $C$ is not enabled in the final state of $\alpha$. 
2. If $\alpha$ is infinite, then $\alpha$ contains either infinitely many events from $C$ or infinitely many occurrences of states in with $C$ is not enabled.

The data exchange agent model has tasks of visiting simulations, querying data, and reporting data to the main simulation. The visiting task is always enabled. For infinitely many visiting tasks, the data exchange agent will have infinitely many occurrences of querying data from simulations and reporting data to the main simulation. Therefore we claim that the data exchange agent model’s execution sequence is fair.

Liveness is the property of the trace of an automaton. An I/O automaton can be viewed as a “black box” from the point of view of user, that is, the user observes only the trace of the automaton. A trace property $P$ consists of the following:

- A signature containing no internal actions: $\text{sig}(P)$
- A set of sequences of actions in $\text{act}(\text{sig}(P)) = \{\text{in}(\text{sig}(P)), \text{out}(\text{sig}(P))\}$: $\text{traces}(P)$

We say a trace property $P$ is a trace liveness property provided that every finite sequence over $\text{acts}(\text{sig}(P))$ has some extension in $\text{traces}(P)$. Let us examine the data exchange agent automaton. As an example, its signature contains inputs $\text{dataIn}$ and output $\text{changeOut}$. Suppose $\text{traces}(P)$ is the set of sequence $\beta$ of $\text{dataIn}$ and $\text{changeOut}$ actions in which, for every $\text{dataIn}$ event in $\beta$, there is some $\text{changeOut}$ event occurring later in $\beta$. Then $P$ is a liveness property for the data exchange agent.
Further exposition regarding the formalism and properties of I/O automaton can be found in Lynch (2000). The data exchange agent is conceptually a state machine in which state transitions are governed by inputs as well as internal activities. What follows is the data exchange agent description in the pseudo code format in Lynch (2000).

**Data Exchange Agent** automaton:

**Signature**:
- **Input:**
  - `requestIn`, the user defined entity states to follow
  - `dataIn`, the data acquired from the state database
  - `order ∈ {round_robin, star, ...}`, the order and pattern of the agent to visit distributed simulations

- **Output:**
  - `changeOut`, the request to change simulation. The simulations respond to this signal and send the agent out as a serializable object
  - `dataOut`, when the agent comes to the base simulation, it uploads data collected through this output
  - `queryOut`, the agent sends queries to the StateRecorder through this port

**Internal:**
- `T ∈ R`, the data exchange agent parameter, the time delay to transfer

**States:**
- `phase ∈ {ready, work, report, change}`
- `location ∈ {base, sim1, sim2, ..., simM}`, the current location the agent is
- `requirement`, a string describing the state information to retrieve

Initially, the agent’s `phase` is `ready`, and `location` is `base`

**msgs:**

- if `(location != base)` then
  - sends query via `queryOut` to `location`’s StateRecorder

- else
  - changes agent phase to `report`;
  - sends data via `dataOut` to `base`’s OutputAnalyst;

- end if

- after T time delay, sends change query via `changeOut` port to `location`
Transitions:

requestIn
Precondition: location = base
Effect: agent sets requirement to input string;
   agent sets order;
   agent phase changes to change;
   agent sends change request via changOut port and is ready to move to next simulation.

dataIn
Precondition: location = sim_i
Effect: agent phase changes to work;
   agent sends query via quearyOut and receives data from StateRecorder;
   agent sends change request via changOut and is ready to move to next simulation.

The inputs for the automaton are requestIn, dataIn, and order, and the outputs are changeOut, dataOut, and queryOut. Input requestIn accepts the user specifications on the simulation model and on the states to follow, which sets the state value of requirement. The dataIn accepts the data from the state database after the data exchange agent sends its queries through queryOut. The order takes values of round_robin, star, etc which specifies the travel pattern of the data exchange agent. When the agent is scheduled to leave the simulation, it sends a changeOut request. The simulation removes the connections of the data exchange agent with other models and sends it out as a serializable object. The output denoted dataOut provides a means by which the agent uploads data to the base simulation. In addition to requirement, the data exchange agent has two more state variables: phase and location. The phase has one of the following values: ready, work, report and change. The location of the data exchange agent is maintained by the location variable. Initially the data exchange agent is at base simulation and the phase is ready. When
it gets a message on requestIn, the phase of the data exchange agent then transforms to change and the agent sends out a message of changing location through the changeOut. The simulation removes the connections between the data exchange agent and other models at the changing request. The data exchange agent then migrates to the designated sub-simulation. When it arrives at the destination, it sets up the connections with other models (e.g. StateRecorder) with the help of the sub-simulation. The phase of the data exchange agent transforms to work state and then sends the query to the state database. After certain simulation time, the phase of the data exchange agent once again transforms to change state to move on. If the data exchange agent comes back to the base simulation, it changes its phase to report and uploads the data through the dataOut.

4.2.4 Software Realization Using DEVSJAVA

One of the more prevalent simulation paradigms is discrete event simulation, which can be conveniently represented using the DEVSJAVA library (Zeigler et al., 2000). DEVSJAVA is a Java-based realization of the Discrete Event System Specification (DEVS) formalism. A DEVS model is a modular system receiving inputs, changing states, and generating outputs over time. Since the I/O automaton can also be interpreted as a discrete event system, it is also convenient to implement in DEVSJAVA.

There are two categories of DEVS models, atomic and coupled. An atomic model directly specifies the system’s response to events on its input ports, state transitions,
and generation of events on its output ports. A coupled model is a composition of
DEVS models that presents the same external interfaces as do atomic models. We
can map the inputs and outputs of the automaton description to the input and output
ports of the DEVSJAVA atomic model. The DEVSJAVA internal and external
transition functions could be translated from the transitions of the automaton model;
and the message function is directly related to the message specification of the
automaton.

The system is designed via a layered architecture (Xu et al., 2004). The lowest
layer is the network layer, which is in charge of the low-level communications
through the network. The simulation applications are the higher layers. They reside
on different computers and exchange simulation entities through the network. The
base simulation is the layer on top of the simulation applications. It is independent of
the simulation applications and interfaces with user and data exchange agent. The
data exchange agent is an autonomous object between the base simulation and
simulation applications to convey the data and perhaps perform some simple analysis
of the data. Some of the advantages of this architecture for data collection include:

1) The base simulation, the StateRecorder and the agent are generic. They are
   independent on the simulation applications;

2) The simulation data are taken by the agent selectively because all simulation data
   may not be necessary for output analysis;
3) It is a scalable system. The data exchange agent only collects the data that are required; therefore it is able to work well in large distributed simulations.

4) It has capability to control the amount of data traffic over the network. In the experiments reported in the following section, we will describe how the system provides such control.

4.3 Implementation and Experiments

The application used in this experiment deals with the distributed simulation of a mine. A mine has loading drives from which ore is shipped, and unloading areas where ore is dumped. The trucks are planned to travel from loading drives to unloading areas. The mine topology is modeled as a graph described by nodes and links. The trucks are modeled as a moving object on the graph, which take pre-planned routes. In our experiments the travel time of the trucks from loading drives to unloading areas is the information of interest to the data exchange agent. One of the important performance indices for distributed systems is the volume of data that is transferred among sub-systems. The trucks not only function as random travel time generators in the simulation, but also are significant contributors to the network traffic. The trucks are modeled as serializable objects and are capable of migrating from one computer to another. The migration of the trucks allows us to investigate the data exchange agent performance under relatively heavy network traffic. Two scenarios of the application are investigated: one with limited interactions between sub-models (Experiment 1) and the other with extensive interactions between sub-
models (Experiment 2). The baseline experiments (Report) that have all the sub-models report the state data for analysis of both models are also investigated for comparisons. The following subsections present the experimental setups and the results of the experiments.

4.3.1 Experimental setup

4.3.1.1 Experiment 1: Simulation with limited interactions between sub-models

In Experiment 1, the graph has six loading drives and three unloading drives, and there are three trucks in the simulation, as shown in Figure 4.2.

The graph is divided into three geographical areas. Each area is modeled as a discrete event simulation and distributed on different computers. Correspondingly,
simulation 1, 2, and 3 are implemented in three computers. Suppose that a truck moves from sub-model 1 (area 1) to sub-model 3 (area 3). Sub-model 1 sends a message to set the flag of the truck in sub-model 3. Thus the truck instance in sub-model 1 is deactivated and the one in sub-model 3 is activated. Each truck has a different speed according to the random number generator in the truck model. In this experiment, most of the inter-model activities happen between sub-models 1 and 3. The data exchange agent is transferred through the network as a serializable object as described in section 2.

The distributed simulation is deployed on the Local Area Network (LAN) connected computers in the Raptors Lab at the University of Arizona. The computers involved in Experiment 1 are named hawk, harrier, lanner and greyfalcon. All of them are Pentium 4 PC’s with Windows 2000 Operating System. The simulation models are represented using the DEVS formalism and are implemented in Java™ 2 SDK. The hawk computer runs the coordinator of the discrete event simulation and the base simulation that is composed of UserInterface and OutputAnalyst. The sub-models are distributed on the other three computers (Zeigler et al., 1999). The communications between the models are realized by the Remote Method Invocation (RMI) provided by Java. The base simulation creates the data exchange agent and passes it to the sub-simulation models. The travel time of the data exchange agent from simulation to simulation is specified by the agent’s parameter called “visiting interval”. It is actually a measure of time for the agent to finish one round of migration specified in simulation time units. A round starts when the data exchange
agent is released by the base simulation and ends when it returns to the base simulation. The time that the data exchange agent takes to finish a round depends on the number of events happening in that period. For different simulation models, the travel time for the data exchange agent can be different. Table 4.1(a) shows the mapping of visiting intervals and the real time intervals in our experiments.

<table>
<thead>
<tr>
<th>Agent “visiting interval” (simulation time per round)</th>
<th>Real time intervals (minutes per round)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.17</td>
</tr>
<tr>
<td>10</td>
<td>0.20</td>
</tr>
<tr>
<td>15</td>
<td>0.22</td>
</tr>
<tr>
<td>20</td>
<td>0.37</td>
</tr>
<tr>
<td>100</td>
<td>1.10</td>
</tr>
<tr>
<td>200</td>
<td>3.83</td>
</tr>
</tbody>
</table>

1(a) Experiment 1: simulation with limited sub-model interactions

<table>
<thead>
<tr>
<th>Agent “visiting interval” (simulation time per round)</th>
<th>Real time intervals (minutes per round)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>3.57</td>
</tr>
<tr>
<td>300</td>
<td>10.29</td>
</tr>
<tr>
<td>500</td>
<td>13.34</td>
</tr>
<tr>
<td>800</td>
<td>18.68</td>
</tr>
<tr>
<td>Infinity</td>
<td>63.00 (simulation finishes)</td>
</tr>
</tbody>
</table>

1(b) Experiment 2: simulation with extensive sub-model interactions

Table 4.1 Data exchange agent’s “visiting interval” parameter and real time interval

The simulation coordinator on hawk controls the execution of the simulations. It asks the distributed models for imminent events and advances the simulation clock to
the next scheduled event. The coordinator also accepts the output from the simulation models and dispatches it to the destination model. If the distributed sub-models exchange information with each other, the sender first sends the information to coordinator. The coordinator parses the header of the message and forwards it to the destination according to the coupling information of the models (Zeigler et al., 1999). Therefore the network traffic mainly happens between *hawk* and other computers. The reason to put the coordinator together with the base simulation on *hawk* is because we assume that the user who controls the simulation is interested in the data and output analysis. However, the coordinator could be deployed on any of the other computers in the LAN. The setup for computer communications is shown in Figure 4.3. The *OutputAnalyst* component is designed to handle the simulation data acquired by the agent with statistical analysis. It offers limited analysis tools such as sample average and sample variance calculations. The software is implemented in an extensible way, so that other analysis tools can be plugged in.

![Figure 4.3 Experimental system](image-url)
4.3.1.2 Experiment 2: Simulation with extensive sub-model interactions

In Experiment 2 the simulation is extended to more complicated version. The graph has four loading drives and four unloading drives and six trucks are involved in complex tasks. The simulation is divided into four geographical areas. The trucks are modeled with greater detail: they simulate the acceptance of commands and are able to solve the contention over the links. The transfer of a truck from one simulation to another is similar to the data exchange agent migration, i.e., a truck is transferred as a serializable object. Detailed simulations are presented in APPENDIX A. Furthermore, there are inter-model activities between all of the sub-models. Because the simulation models have more details, there are more events generated than in Experiment 1. The difference can be inferred from the data exchange agent’s “visiting interval” parameters, which is highly correlated to the number of events per unit time. The comparisons of the visiting interval parameter and real time interval are given in Table 4.1(b). As an example, for visiting interval parameter being 100 for both Experiment 1 and 2, the real time interval for Experiment 2 is about 3 times of that for Experiment 1. More network traffic is also observed in this experiment due to the transfer of trucks and data exchange agent.

The deployment of the distributed simulation is similar to that of Experiment 1. There is one more computer named lagger that is used in the experiment. Sub-model 4 runs on this computer; all other computers have same role as in Experiment 1.
4.3.1.3 Baseline Experiment: A Centralized Approach

Another way to keep track of model states and the event list is to have all the distributed simulations report data to the user. This mode is referred as “centralized approach”. In this approach, the data extraction and analysis are accomplished at the main simulation by the user. In this case, the network traffic consists of not only information exchanged between the sub-models but also the state and event data reported to the main simulation. Our experiments also explored the performance of this design based on the network traffic generated. The advantage of this design is that although data are generated in different simulations, the main simulation and user see a centralized integrated database. However, reporting all data through the centralized approach can cause significant increase in the network traffic.

4.3.1.4 Running the Experiments

For all simulation models, five sets of random number seeds are used to perform the experiments independently. The network traffic data observed are averaged over the five runs.

In order to obtain the traffic volume data of the network, we used windump utility to monitor the number of packets passing between the computers. Windump is the porting of tcpdump (the most used network analyzer for UNIX) to the windows platform (http://netgroup-serv.polito.it/windump/). The Windump data file has entries of the following form:
which means at 10:57:10.079596, *hawk’s* port 7002 sent 1 tcp packet to *harrier* at port 1555 and the total tcp packets so far were 266. Reading through the windump data file, we are able to calculate the total number of packets between the computers. In order to avoid having other communications programs introduce packets into the network, we ensured that no other communication software was running during the experimental procedures.

### 4.3.2 Results and Analysis

In Experiment 1 and Experiment 2, the network traffic data include the inter-simulation communication such as the transfer of the vehicle from one area to another; the migration of the data exchange agent; and the data that the agent carries. If the simulation applications are same, the network traffic data caused by the messages among them remain unchanged. However different migration frequency of the data exchange agent can cause different network traffic. The experiments illustrate how the agent's “visiting interval” parameter affects the network traffic. The results can help adjust the “visiting interval” parameters with respect to the tolerable amount of network traffic.
Figure 4.4 Average total packets transferred between simulations

(a) Experiment 1: simulation with limited simulation interactions

(b) Experiment 2: simulation with extensive simulation interactions
Figure 4.4 (a) and (b) show the overall tendency of the frequency of the agent migration and packet volume for Experiment 1 and 2 respectively. There is not much variance for the data and therefore the average of total packets being sent among the computers is shown in the figures. The less frequently the agent migrates from one simulation to another, the less data are transferred between the simulations. This is because the agent is less likely to transport unanalyzed data while moving at lower frequencies. When the agent moves very slowly, the data carried by the agent and the agent size are not very significant. The information exchanged between simulations is more significant to the network traffic. Therefore, the total number of bytes converges to the total information exchanged between simulations (about 40 Mbytes for Experiment 1 and 550 Mbytes for Experiment 2). From Figure 4.4 we can see that the “visiting interval” parameter should be chosen to avoid operating in very sensitive regions. Thus in Figure 4.4 (a) parameter settings corresponding to frequencies near 1.0 minute/round are robust, whereas those with very small frequencies increase packet volume dramatically. The same results are observed in Figure 4.4 (b). These experiments also provide guidance for how to choose Data Exchange Agent’s visiting interval parameter. Suppose that there is a limit of 70 Mbytes network traffic for Experiment 1, a reasonable “visiting interval” parameter is around 30, which results in 0.5 minutes per round. If the network traffic limit is 700 Mbytes for Experiment 2, a reasonable “visiting interval” parameter is around 300, which makes 10.20 minutes per round.
Figure 4.5 (a) and (b) report the average total traffic for Experiment 1 and 2. It is possible that if the data exchange agent’s visiting interval parameter is not chosen properly, the traffic is more than the “centralized approach”. Figure 4.5 (a) shows that the centralized approach has less traffic volume than agent design with parameters 5.0 and 10.0, but has more traffic than the agent design with parameter larger than 15.0. This observation is not shown in Experiment 2, because the parameters chosen are big. It is conceivable though, if the parameter is small, the report design might have less traffic volume, because transferring the data exchange agent becomes dominant. There are several factors to consider in deciding the parameters, which include the data exchange agent’s size, the throughput of the simulations and the speed of the networks, etc. Experiments show an estimation of the data exchange agent size without carrying any data to be 730Kbytes.

Figure 4.6 (a) and (b) present the average traffic volume from computer to computer (simulation to simulation). The data also support that the agent system is able to reduce the network traffic volume if the data exchange agent’s visiting interval parameter is chosen properly.
Figure 4.5 Average total traffic: agent system v. centralized approach

(a) Experiment 1: simulation with limited simulation interactions, the X-axis is data exchange agent visiting interval parameter

(b) Experiment 2: simulation with extensive simulation interactions, the X-axis is data exchange agent visiting interval parameter
4.3.3 Class Diagram

The class diagram of the system is shown in Figure 4.7. This class diagram only shows sub-simulation 1 and the main simulation, because other sub-simulations are similar to sub-simulation 1. Each model component works as a remote server to provide methods to run the sub-simulation, whereas the RMIClient (coordinator) invokes these methods and controls the procedure of the distributed simulation.

4.4 Summary

This chapter introduces an agent-based data collection architecture for distributed simulations. An experimental example arising within a mining application is used to illustrate the implementation. Comparisons of the network traffic volume according to different agent travel frequencies and the “centralized” approach are presented. If the parameters are chosen appropriately, the agent-based data collection architecture provides statistical analysis without adding much network traffic volume. The future research will concern with providing more intelligence to the data exchange agent. The data exchange agent should be able to perform some more complicated tasks. The agent can adjust the “visiting interval” parameter according to the situations of the data collection. The agent may also be able to find an optimal route to visit the sub-simulation, or the data exchange agent can detect whether a sub-simulation has gone awry from the data it acquired, and so on. In addition, there are cases in which
each distributed simulation may store data in a customized database that is specific to a sub-model. In this case an intermediate agent may be necessary to transform the customized data format a common data format so that the data exchange agents can extract information.
Figure 4.6 Average total traffic between computers

(a) Experiment 1: simulation with limited simulation interactions, the X-axis indicates the computers involved in the communication

(b) Experiment 2: Simulation with extensive simulation interactions, the X-axis indicates the computers involved in the communication
Figure 4.7 Multi-agent data collection system class diagram
In this chapter, we demonstrate that the SPEED-CS architecture can be used for parallel reality systems. A parallel reality system is a hybrid system that includes software models and physical models interacting with each other. Parallel reality systems are widely applied in areas to model and control dynamic systems, or to train human behaviors. In parallel reality systems, physical models can be used as data sources for software models. We implemented the framework under the SPEED-CS architecture, which is able to collect the data of the physical models and of the software models, and develop a mechanism to update the software models’ parameter adaptively. This mechanism helps the software models maintain the fidelity of the simulation. The framework is a multi-agent based architecture, in which “data exchange agents” manage the collection, dissemination, and analysis of data from dynamic data sources including simulations and/or physical systems. The framework utilizes XML as a common data model for information exchanged among systems. We demonstrate the applicability of the framework through an experiment involving simulation of miniature autonomous vehicles in a laboratory setup which we refer to as “open bit mine.”
5.1 Introduction

Simulation modeling has become one of the mainstays of the modeler’s toolkit. Its usage covers a broad spectrum of applications including communication, healthcare, manufacturing, military, transportation, and many more. Many of these applications are reported in various simulation journals and conferences (Price and Harrel, 1999; Schaefer, 2001; Hill et al., 2001; Warren et al., 2004). The technology for developing simulations includes a variety of modeling languages such as Arena (http://www.arenasimulation.com), ProModel (http://www.rapidmodeling.com), DEVS (http://www.acims.arizona.edu/SOFTWARE/software.shtml), input and output analysis tools such as BestFit (http://www.palisade.com/html/bestfit.asp), etc. Many of the software products also provide tools for animation, and visualization. Over the past decade, advances in software technology have led to a fairly mature set of tools for developing simulation models. Nevertheless, several software (and modeling) challenges remain. One such challenge is the ability to maintain model fidelity (Baileg and Kemple, 1992). For instance, parameters of a simulation model are often determined through an a priori statistical analysis. Once the model has been developed, these parameters are usually updated periodically. However such updates are performed only when an analyst recognizes that the parameters may be out-of-date. If the simulation model is exercised by an inexperienced user rather than a developer/analyst, the detection of out-of-date parameters may go unnoticed for a while. Correspondingly, simulation-based decisions and control may be “out-of-
touch” with reality during such periods. In this chapter, we propose a “parallel reality” simulation framework, which is a software tool that helps a simulation model stay synchronized with reality without constant oversight from the users/analysts.

In addition to the above role, a parallel reality simulation will also provide a mechanism by which models of different scales can inter-operate within one environment. In modeling complex systems, certain components of a model involve high degree of abstraction while others may need greater detail. Such disparities in detail may arise in cases where certain technologies may be very well understood while others may need exploration in greater detail. The different levels of fidelity, allowing alternative hardware and software configurations, alternative operating policies etc. lead to a simulation model that operates “parallel realities.”

In order to give readers a glimpse of the types of applications that motivate this work, we provide a few scenarios in which a parallel reality simulation is appropriate. It is important to note that the following examples cover a wide range of cases. It is this generality that makes the parallel reality framework novel.

• (“Hardware in the loop.”) Consider a scenario wherein an autonomous vehicle is to be introduced into an operation in which other vehicles are operated by human operators. Since the autonomous vehicle may be an emerging technology (say in mining operations), its performance may be uncertain, and hence difficult to model well. It is most reliable to allow this new vehicle to be represented as a real vehicle within a simulation while other human-operated vehicles are
represented within a virtual world. Thus, during a time-interval when the mine is
shut down, we may run a parallel reality simulation in which data regarding
human-operated vehicles are used in an abstract (virtual) world, whereas, the
newly acquired autonomous vehicle is operated in the “real-world.” It is
important to note that because the mine is not in operation, the autonomous
vehicle does not encounter the same level of hazards as in the actual operation.
Yet, by interfacing the new equipment to the parallel-reality simulator, we are
able to investigate and study potential problems with introducing new technology
into the operation. A parallel reality model can be used here to simulate virtual
vehicles (driven by human operators) with real autonomous vehicles as if they are
both running on the same road-network. Moreover, data from the real
(autonomous) vehicles can be used to gather information regarding its operations.
Once significant data about the new technology have been obtained, its
performance can be modeled in future simulation studies.

• (“Human-in-the-loop”.) Consider an application in which students in finance are
trained as financial planners through the use of a simulation in which some
investors are real (e.g. other students), but other clients are simulated
representations so that they can gain experience in dealing with a variety of risk-
preferences. In addition, the stock-market data can also be authentic. In this
example, there are several sources of data, some of which are real (the stock-
market and some investors), whereas, others are generated from simulations.
• (Modeling Dynamic Systems.) Modeling the spread of forest fires in real time is a challenging task (Wilson et al., 2001). A forest fire is a highly dynamic system, which requires control strategies that respond dynamically to changes. When a forest fire occurs, simulations are utilized to determine the likely progress and directions the fire may spread. The simulation results provide command centers with information to assign resources. Periodically, current data must be fed back into the simulation to keep the fidelity of the models. Some small, easily deployable sensors can be used to gather the real-time measurements (Corr and Okinno, 2001). Through the parallel reality framework, the related parameters in the simulation can be updated to provide information corresponding to spread of the fire.

• (Controlling Dynamic Systems.) Modern traffic control methods such as COP (Sen and Head, 1997) use sensor information together with a traffic model to determine optimal signal control. The traffic model itself is a simplified simulation model with parameters such as turning probabilities being used to create realistic traffic patterns. These parameter estimates may be updated with real-time sensor data using the parallel reality framework suggested in this chapter.

The applications mentioned above are very difficult to model using standard simulation technology (especially current simulation languages). While each simulation can be developed using special-purpose programming, such an effort suffers from a disadvantage that each of these models have to be developed “from
scratch”. Every scenario requires an entirely new planning and startup effort, with software from the previous developments being difficult, if not impossible to use. The basic aim of our project is to provide a framework, which reduces software costs by facilitating the reuse of simulation components and by providing a runtime infrastructure to manage the simulations.

The organization of this chapter is as follows. In Section 2, the general framework of parallel reality is discussed. We outline the basic features and functionalities that are necessary for the framework to possess. In Section 3, an example application involving physical and virtual vehicles is described within the semantics of the general architecture. While much of this presentation is generic (and transferable across applications), some aspects of the architecture allow specialized modules. We illustrate these aspects through a discussion of software modules that are associated with the illustrative application presented in this chapter. Important issues as synchronizing virtual time with real time are also discussed. Section 4 provides experimental results, and in section 5, we discuss future research directions.

5.2 Parallel Reality Framework

5.2.1 Distributed Simulations and Dynamic Data Sources

Data accuracy is a critical factor for simulations to be able to produce accurate results. The data sources may vary for different simulation models. Some simulations use data generated by other simulation modules, whereas, others use real-
time data with which the simulation models interact. These data sources provide dynamic data, which can change from time to time. The objective of the parallel reality framework is to provide modelers with architecture that is able to integrate simulations with asynchronous dynamic data sources, particularly real-time data sources.

In order to develop the parallel reality framework that can effectively utilize dynamically evolving real-time data sources, distributed discrete-event simulation provides some of the required technologies. Discrete-event simulation has become accepted as one of the main tools for developing simulation models (Law and Kelton, 1999). In parallel reality, the introduction of real-time data in the simulation will trigger discrete events, which will be referred to as “Hardware Events” in order to differentiate them from events generated by other simulation events that are called “Software Events”. Consequently, the software modules developed for handling these two categories of events are called “Simulation with Hardware Events” and “Simulation with Software Events” respectively. Integrating these two types of event using distributed discrete-event simulation will allow us to facilitate simulation methods to avoid possible causality errors caused by different clocks.

5.2.2 Multi-Agent Based Systems

Software agents provide flexibility and efficiency to coordinate communication among different modules of a simulation. A software agent is an autonomous computer program that operates on behalf of someone or something. If a software
agent can migrate under its own control within a computer network, the agent is a mobile agent. If it remains on the system on which it began execution, it is a stationary agent. A mobile agent demonstrates many advantages in distributed computing systems. It can reduce the network traffic between simulation models by extracting and compressing data. It can adapt to changing migration paths depending on its task and current network conditions. A mobile agent can be suitable for a distributed simulation setup over heterogeneous platforms, because a mobile agent is designed to be system-independent (Xu et al., 2004). The parallel reality framework requires both mobile and stationary agents. The multi-agent system not only is efficient for handling large volumes of data, but also can provide flexibility and scalability for modules with differing complexities.

5.2.3 Functionality of Parallel Reality Framework

The parallel reality framework is an interoperable system allowing both hardware and software sub-systems. It integrates data obtained from different sources and analyzes these data to update the software simulations with more accurate parameters. A hardware system consists of physical objects and sensors. The software system not only consists of modules that are independent of physical objects, but also requires software representations of the physical objects in a software simulation. The software representations of the physical objects are modeled as simulation entities that are characterized by state variables and state transitions. State transitions are often governed by random variables for which probability distributions are developed through data collection and statistical input modeling.
The parallel reality framework is designed to automate the effort of maintaining the simulation fidelity adaptively. It keeps track of two properties of the entities within the Hardware Simulation: the state transition and the time when the transition occurs. These observations are stored in a database called the Real Event State Database because the events record state transitions in the physical reality. The state transition and the transition time of the entities in the Software Simulation are stored in the Simulated Event State Database. For an entity within either Hardware Simulations or Software simulations, the durations of an entity staying in a certain state can be acquired by Data Exchange Agents (Xu et al., 2004). As more data becomes available, statistical estimates (e.g. parameters of distributions, moments etc.) can be updated so that the simulation begins to adaptively learn the behavior associated with the physical world.

Parallel reality framework has some similarities as well as differences with virtual reality research. Virtual reality has long been used in areas where virtual environments help humans coordinate their actions, such as rehabilitation devices used in clinics and researches (Girone et al., 2000), the virtual football trainer (http://www-vrl.umich.edu), and so on. Both parallel reality and virtual reality deal with virtual and real environments. While virtual reality focuses more on using simulation to change reality (such as improving human responses), parallel reality focuses more on using the real and virtual world data source to better mimic the reality and to make more reasonable decisions based upon the simulations.
5.2.4 Architecture of Parallel Reality Framework

The general architecture for a parallel reality framework is shown in Figure 5.1. The dotted boxes indicate that the whole system is composed of two types of simulation systems: a Software System and a Hardware System. Within the Software System box, there are two components to handle software event simulation and real event simulation respectively. *Simulation with Software Events* module includes a discrete-event simulation model of the systems that interacts with the hardware systems as well as discrete-event simulation of the Hardware systems. Data of state transitions of simulation entities are stored to *Software Event State Database*. *Simulation with Hardware Events* module is a discrete-event simulation that interfaces with the real Hardware System. It responds to events generated from the Hardware System and reports these events to *Real Event State Database*. The *Simulation Coordinator* integrates these events with the *Simulations with Software Events* and maintains the causality of the simulation systems. This is the key component that monitors the time advancing mechanism and message passing among simulation models. The *Simulation Coordinator* can also send the *Hardware System* commands to adjust the behavior of the physical objects.

In order to extract information from the databases, several *Data Exchange Agents* may be deployed to extract specific information from the State Databases (Xu et al. 2004). Such data exchange agents are able to communicate with modules known as *Output Analysts*, which in turn perform the appropriate statistical analysis. The analysis can provide the simulation with new set of parameters. Note that one can
determine whether the parameters used within a *Simulation with Software Event* are within acceptable tolerances using methods from statistical process control. Such methods of model validation become possible due to the parallel reality framework.

![Figure 5.1 Parallel reality framework](image)

The *Hardware System* consists of two sub-systems: *Actuator System* and *Sensor System*. The *Hardware System* can be a hardware experimental system of the reality or even the reality itself. The *Sensor System* collects real-time events in the Hardware System and transfers these events to Software System. The *Actuator System* can be used to execute the commands that are generated from the simulation decisions. The implementations of the Hardware Systems vary with the real world applications to be simulated. More details will be provided as we illustrate the experimental system of the parallel reality framework with a simple example of a traffic control system.
5.3 An Experimental System

5.3.1 Overview

We have developed an agent-based software framework to collect data between different simulations in chapter 3. The framework is part of the SPEED-CS architecture. The framework applies mobile Data Exchange Agents to travel over the network and visit different simulations, and to search for data that simulations may need. We have shown that for systems that generate a large volume of data, the Data Exchange Agents reported in (Xu et al., 2004) reduce data traffic over the network significantly. We show that the framework can be used to develop the parallel reality system as well. An experimental implementation is developed within the SPEED-CS architecture. It demonstrates the ability of architecture to accomplish the data sharing among dynamic data sources on heterogeneous platforms. It also shows that simulations do not need to have the knowledge of each other, but the agents must agree on the data format and protocols to make communications between simulations possible.

A common data format is required to possess the following basic attributes (Staniford-Chen, 1998):

- **Self-descriptive.** It should be self-evident from a message how each datum within it should be interpreted.

- **Layered.** The layered structure is suitable for expressing data with structural relationships.
• *Extensible*. There should be a mechanism such that user-defined vocabularies are allowed to add into the data streams.

It is not a trivia task to develop a common data format. Fortunately the eXtensible Markup Language (XML) satisfies all the aforementioned requirements. XML is a simple, yet flexible text format. Originally designed to meet the challenges of large-scale electronic publishing, XML is playing an increasingly important role in the exchange of a variety of data over the computer network. In our framework, XML strings are the basic data format for information representation.

We have designed an application that mimics ore-pick-up and drop-off in open-pit mining. Since it is a laboratory-based application in which the ore to be transported is a “bit” of information, we refer to this application as an “open-bit” mine. While load/haul/dump (LHD) vehicles in an open-pit mining system travel from loading zones to dumping zones to transport ore, our open-bit mine uses autonomous vehicles (robots) that travel from sources to destinations to transport just a bit of information. The experiments have both hardware/physical as well as software/simulated vehicles. In order to simulate the productivity of the system, one of the more significant parameters is the estimated travel time to complete one trip. We will use the parallel reality framework for the simulation, and demonstrate that the proposed framework can be used to dynamically update parameters of simulation entities. Our implementation for this example includes general-purpose software components, which are required for the generic parallel reality framework. These components are the *Simulation Coordinator*, *Event State Databases*, *Data Exchange Agent*, and
Output Analyst. The example also includes components directly related with the open-bit mining experiments. Finally, we provide a brief description of a user-friendly graphical user interface (GUI) in the Appendix B.

The simulation components follow the DEVS formalism (Zeigler and Sarjoughian, 2003a). We implement the software components using DEVSJAVA package.

5.3.2 Implementation of General-purpose Components

General-purpose software components form the basic structure of the implementations of the parallel reality framework. These components are required for all applications. They provide services to coordinate distributed simulations, to manage discrete events, to transport and analyze data from different simulation, and to interface with the physical devices (sensors and actuators).

5.3.2.1 Simulation Coordinator

The Simulation Coordinator component monitors the execution of the discrete event simulations of each simulator. Figure 5.2 illustrates the parallel DEVS simulation protocol (Zeigler et al., 1999). The Coordinator queries all clock times of next event from each simulator component. It sends the minimum value of these times back to the components, thereby allowing them to determine whether they are imminent, and if so to generate output. More than one component may be imminent and the outputs of all these components are sorted and distributed to others according to coupling rules.
The coordinator is able to provide distributed simulation services with this protocol. Simulators can be deployed as server programs over the computer networks, and they send information to the coordinator remotely. The Hardware System can join the simulation pool as long as the software interfacing with the Hardware System follows the DEVS protocols.

![Diagram of Parallel DEVS simulation protocol](image)

**Figure 5.2 Parallel DEVS simulation protocol**
5.3.2.2 Event State Databases

In discrete-event simulations, events cause the state transitions of entities. The state transitions and the times of the transitions allow the parallel reality systems to obtain the time durations of which an entity stays in a certain state. These time durations can provide us the data to perform statistical analysis for some parameters. Therefore we record the state transitions and the timestamps of the transitions of entities in the Event State Databases.

Events can be generated from different sources. They can come from simulation entities within software simulations. The state transitions and the transition times caused by this type of events are stored in the Software Event State Database. Events can also come from the Hardware System. For these events, the state transitions and the transition times are stored in the Real Events State Database. Since the state transitions are used to describe the real events, CPU clocks are stored in the Real Event State Databases as the transition times.

5.3.2.3 Data Exchange Agent and Output Analyst

A data exchange agent is a mobile software agent whose task is to visit Event State Databases and to obtain time durations of a specified state of a simulated entity. The design issues of a data exchange agent include (Xu et al., 2004):

- *The tasks to be accomplished.* A data exchange agent is designed to search and collect simulation information.
- **The realization of mobility.** A data exchange agent needs to be able to migrate from one simulation to another simulation over the network. It should be able to make network connection to the ongoing simulations.

- **The information format.** A data exchange agent must agree on certain data format so that it can extract information.

- **The intelligence level.** A data exchange agent should have the intelligence of making decisions whether the data collected is sufficient, or whether the data contains too much noise and whether longer data streams may be necessary. If multiple data sources provide measurements of the same object, the data exchange agent needs to be able to decide how to reconcile any differences.

The **Output Analyst** is a stationary agent to process the data and to extract the statistical attributes of data. A simple set of the statistical attributes may include parameters such as mean and variance of the durations of an entity staying in a certain state. These parameters help the parallel reality framework monitor the simulated entity against the physical object. If the parameters of the simulated entity deviate too much from those of the physical object, the **Output Analyst** will inform the simulation to update the parameters of the simulated entity. The **Simulation Coordinator** then immediately schedules the event to update the parameters of the simulated entity, so that the simulation can be adapted to be more accurate.
5.3.3 Application-Specific Implementation

5.3.3.1 Open-bit mine

In this section, we present the software components that are necessary for studying the performance of an “open-bit mine.” As mentioned previously, an open-bit mine is a laboratory caricature of an open-pit mining system. Accordingly, many of the software modules mimic similar modules in open-pit mines. These include vehicles (including controllers), planning systems, sensing systems, and actuating systems. In order to demonstrate the value of the new parallel reality framework, we focus on modeling the vehicle travel times from source to destination in the simulation. To do so, a set of simulated vehicles is first modeled with the data collected from observations of the hardware (vehicles) before the parallel reality experiments. The paths on which the vehicles travel are modeled with nodes and links in the simulation. We refer to the model of the paths as virtual paths. Simulated vehicles travel on virtual paths, and the hardware (vehicles) travel on physical paths that we setup in the laboratory.

Figure 5.3 shows the components that set up this experimental system. The system is constructed upon a local area network (LAN) and a wireless network. Software models are distributed over the local area network. The wireless network is the communication mechanism for the hardware vehicles to send and receive data from the coordinator. Each component of the experimental system will be discussed in greater detail below.
5.3.3.2 Planning System

The planning system consists of two components: PathPlanner and TaskPlanner. The PathPlanner implements path-planning algorithms according to a given roadmap represented as a graph with nodes and directed links. The path planning algorithms are user-defined algorithms to make decisions on the routes for the vehicles to take. The source and destination of the vehicles are user inputs. We have designed a common interface AlgorithmInterface and in this common interface the getPath() method is implemented based on different algorithms to generate a
vector of links. These links are connected to form a path from the source to the designation. The PathPlanner component will retrieve this path for simulations.

Two algorithms are implemented for demonstrative purpose: they are Dijkstra shortest path algorithm and a congestion control algorithm. Both implementations instantiate the `getPath()` method to generate path from source to destination. The Dijkstra implementation applies the standard Dijkstra algorithm and the estimated delays are taken into consideration at the planning stage. In congestion control algorithm, a link can be reserved for the use of a simulated or hardware vehicle until the vehicle finishes the trip on the link. It plans based on the fact that a link can be reserved only by one vehicle at any time and therefore considers the delay caused by the link unavailability. More sophisticated algorithms can be developed for this problem, but it is out of the scope of this experiment. In this chapter our focus is on the operation of the simulation within a parallel reality framework.

```
<path truckId="1" tripNum="1" speed="high">
  <node id="1" x="0.0" y="0.0">
    <expectedT>0.0</expectedT>
  </node>
  <node id="2" x="0.0" y="30.0">
    <expectedT>40.0</expectedT>
  </node>
  <node id="3" x="0.0" y="115.0">
    <expectedT>130.0</expectedT>
  </node>
</path>
```

Figure 5.4 An example of XML output schema from PathPlanner
The *TaskPlanner* is designed to parse the path information provided by the *PathPlanner*, and provide each vehicle the next task it needs to perform. The *PathPlanner* outputs the results of the algorithm with a stream of XML string. An example of the results from the *PathPlanner* is shown in Figure 5.4. This XML string defines a path of the first trip for the vehicle with “truckId” 1. The vehicle is required to start from the node with Cartesian coordinates (0.0, 0.0) at time 0.0. It is expected to arrive at node (0.0, 30.0) at minute 40.0 and then to arrive at the destination node (0.0, 115.0) at minute 130.0. As we can see the path designed for this vehicle is a straight line with two links. There is another important piece of information encoded in the string: the expected speed range of the vehicle. The speed range of the vehicle can take one of the three values: high, medium, or low, because the hardware vehicles have these three settings for speed. A speed range will point to the statistical mean calculated from the observed speed data when a hardware vehicle is set to the speed range. Note that these data may introduce randomness in the simulation.

```
<commandInfo>
  <command>1S025</command>
  <pathAngle>1.57</pathAngle>
  <estStartT>0.0</estStartT>
  <estEndT>40.0</estEndT>
</commandInfo>
```

Figure 5.5 An example of command information
Each vehicle used in our simulation can only accept simple commands. Because a vehicle is driven by a small step motor, it takes commands of direction together with number of steps for the motor to run. The TaskPlanner needs to convert the paths received from the PathPlanner to the commands that the vehicles accept. It also needs to obtain the expected times at which a vehicle is scheduled to arrive at a node. These expected times give the Scheduler guidelines for the sequence of the commands. In order to decide the direction in which the vehicle is supposed to go, it is necessary to include the angles of the link. The data shown in Figure 5.5 is the command information extracted from the first link of the path example. The command “1S025” indicates that the vehicle with id 1 needs to go straight (S) for 25 steps on the link. The command string is composed of three parts: the vehicle id number, the direction, and the number of steps. The direction can take one of the following characters: “S”, “B”, “L”, and “R”, which stands for moving straight, moving back, turning left, and turning right respectively. The number of steps is a 3-digit integer for the step motor. The angle of the link is 1.57 (\(\pi / 2\)) with respect to horizontal axis in a coordinate system. This path angle requires the vehicle to go along the direction as the vertical axis points. The vehicle is expected to start at time 0.0 and to arrive the node at minute 40.0.

For the vehicle to accomplish the planned path, the command XML string may contain multiple commands generated by the TaskPlanner. The outputs are then forwarded to the Scheduler.
5.3.3.3 Scheduler

The Scheduler component receives the input from the TaskPlanner and sends the command out. To schedule the time to send out commands, the Scheduler compares the expected start time with the simulation clock time. If the current simulation time is earlier than the expected command starting time, then the Scheduler will put the event of sending out the command in the event queue and schedule this event at the expected starting time. Otherwise if the command has already been delayed, the Scheduler will send out the command as soon as it is able to do so without violating causality of the system. The commands are sent to the processors of either the simulated or the hardware vehicles. The simulated vehicles are modeled such that it mimics the execution of the commands by a hardware vehicle. More details about the hardware vehicles will be presented in section 3.3.5.

5.3.3.4 Control Switch

The Control Switch is the component connecting hardware vehicles and other software components. It is a DEVS atomic model that controls the process thread of sending the commands to the hardware vehicles. The vehicle id in a command string indicates whether the command is for a hardware vehicle or a simulated vehicle. When a command for the hardware vehicle is injected to the Control Switch, a process thread is generated for the transmission to the designated vehicle. The transmission follows a wireless protocol and is facilitated by a transceiver IO card in the computer. There is a manufacturer-provided dynamic link library (dll) to offer APIs for our program to accomplish the transmission. We have developed a C++
program to implement this task. Because the simulations are modeled in DEVSJAVA, we use Java Native Interface (JNI) to call the C++ program from the Control Switch. The JNI allows Java code that runs within a Java Virtual Machine (VM) to operate with applications and libraries written in C/C++ (http://java.sun.com/j2se/1.4.2/docs/guide/jni). The transmission thread is destroyed when the transmission ends.

5.3.3.5 Hardware Vehicle (Actuator System)

A hardware vehicle is composed of a communication interface, driver software for the step motor, and a microprocessor with a small memory. The communication interface is used for the vehicles to accept commands transmitted from the Control Switch. The manufacture-provided wireless protocol establishes the connections between the computer and the interface. The driver software runs the step motor with the direction and steps as the input parameters. These parameters are extracted from the commands given by the Control Switch. The driver software resides on the memory of a microprocessor on the hardware vehicle. In the design of the hardware vehicles, we have used two different microprocessors. For the older models, we use BASIC STAMP II processor (http://www.parallax.com); for the newer models, we use TINI processor (http://www.emacinc.com). The TINI processor is a JAVA programmable processor and therefore allows us to set up the interoperability between the vehicle and the simulations easily.
5.3.3.6 Sensor System

The sensor system is a key component that provides the simulations with the information about the physical reality. An efficient sensor system should give the simulations accurate and up-to-date data about the objects to be simulated. In our experiment, we are interested in the time durations that a hardware vehicle takes to move on a link. The positions of the vehicle indicate whether the vehicle has finished the trip on the link or not. In the experiment, we assume that each link has a capacity of only one vehicle and that each node has the capacity of more than one vehicle. Therefore if there is a hardware or simulated vehicle on the link, the link is not available until the vehicle has arrived at a node. With the positions of the vehicles, the Scheduler can decide whether the vehicle can be commanded to start the trip on the link safely, or the vehicle has to wait for the link to be available.

In our experiment, we set up a simple IRAD sensor at each node of the graph. Because each link allows only one vehicle, this set up is sufficient. For more complicated physical systems, more sensors are required. A network of sensors is constructed and each of the sensors has a unique identity number. The network is connected to a computer. If a sensor is activated, it will send a pulse to an IO register on the computer. The computer is able to detect the id number of the sensor and then sends the number to the simulations. The Control Switch component generates a thread listening to the specific port for the wireless communication. After it receives the data, it informs the Scheduler to schedule this event as the next imminent event. Because the simulations have the data of the positions of the sensors in the coordinate
system, they can also infer the current position of the vehicle that has triggered this sensor. This event can cause the state of the vehicle to change. The new state as well as the current CPU time is logged into the Hardware Event State Database. A schematic diagram of the open-bit mining experimental system is shown in Figure 5.3.

5.3.4 Data Processing and Parameter Updating

A vehicle is designed to be at one of the three states: “unloading”, “running”, and “loading”. The state transitions are related to the events. For example, a vehicle is initially at the source node and its state is set to be “loading”. As soon the vehicle receives the commands of moving along the path, the state of the vehicle changes to “running”. When the vehicle reaches the destination node, its state changes to “unloading”. For the hardware vehicle the sensor at the destination node will send an event to the Simulation Coordinator indicating that the state of the vehicle has changed. For the simulated vehicle its coordinates on the graph will reflect the position of the vehicle. If the position of simulated vehicle is close enough to the destination node, the simulation indicates that the vehicle is at the destination node. The event of arriving at the destination is then generated and reported to the Simulation Coordinator. These state transitions and transition times are inserted to the Event State Databases. We can obtain the duration of the vehicle keeping in a certain state by calculating the difference between the time of the vehicle entering to the state and the time of the vehicle exiting the state. For example, the time difference between the “loading” state and the immediate “unloading” state of a
vehicle is the duration of the vehicle being in the state of “running”, which is the travel time and waiting time of the vehicle from the source node to the destination node.

*Data Exchange Agents* are responsible for collecting the data from the *State Databases*. The *Output Analyst* performs statistical analysis for the collected data to obtain estimates of the parameters. These parameters are then used in the simulated vehicles to mimic the behavior of the hardware vehicles. During the simulation, data exchange agents are released to migrate the *State Databases* to obtain the time durations between two specified states. In our example, running time, which is the time duration between the state of “loading” and “unloading”, is of interest. The data exchange agent visits the *State Database* and queries for states and time of “loading” and “unloading” once every certain interval that is defined by the “visiting interval” parameter (Xu et al. 2004). Then the data exchange agent reports the data to *Output Analyst*, which will process the data and obtain multiple data points (samples) of the running time. Adequate samples will allow *Output Analyst* to run statistical analysis and to yield improved estimates of the running time. Because the data exchange agent visits both *Real Event State Database* and *Software Event State Database*, the analyses will produce two sets of the parameters for both hardware vehicles and simulated vehicles. The differences of the two sets of parameters will help *Simulation Coordinator* monitor the simulations against the reality. If the set of parameters of the simulated vehicles deviates from the ones of the hardware vehicles,
the *Simulation Coordinator* will update the set of parameters of the simulated vehicles with that of the physical vehicles.

5.3.5 Class Diagram

Figure 5.6 shows the class diagram of the parallel reality experimental system. In the model, the real system communicates with the simulation system via the coordination of RMIClient (*Simulation Coordinator*). The SenserServer class is an independent thread, which keeps listening to activities of the sensor system. Object *SensorModel* extends from the atomic class in DEVSJAVA and thus can be coupled into the simulation system.
5.4 Experimental Results

An open-bit mine experimental system, which deals with the interoperation of the hardware vehicles and simulated vehicles, is designed to demonstrate the concept of the parallel reality framework for simulation and modeling. In the specific experiments there are two vehicles to travel on a straight path that has sensors located at both ends. One of the vehicles is a hardware model, and the other one is a
simulated vehicle. Initially, the traveling times of the simulated vehicle are modeled as uniformly distributed random variables with parameters acquired from the prior knowledge about the speed of the hardware vehicle, plus estimated delay that may be caused by the path unavailability. The prior knowledge is obtained in separate experiments, in which the speeds of a hardware vehicle are collected and analyzed.

The travel time is the key index for the accuracy of the experimental model. As in the experiments, the travel time intervals include possible waiting time at the nodes, which will introduce more uncertainty. A data exchange agent collects the traveling time intervals of the simulated and hardware vehicles from the State Databases. After each trip the mean of traveling time intervals of the simulated vehicle is updated and is set to be the moving average of the current real vehicle travel time and the previous data. Since we assume the uniform distribution of the travel times, we keep the spread between the maximum and minimum to be a constant. In our experiments, the data exchange agent is set to visit the State Databases every 2000-simulation clock unit.
Figure 5.7 shows the experimental results. Both the simulated and hardware vehicles are designed to run 10 round trips on the path. The hardware vehicle and simulated vehicle share one link and when the travel time of the simulated vehicle is observed the waiting time for the link availability is included. For the hardware vehicle however the waiting time for the simulated vehicle to finish a trip in CPU time is very small. The hardware vehicle starts with the first trip. In the first trip, the estimated travel time of the simulated vehicle is not very accurate compared with that of the hardware vehicles. The simulated vehicle’s travel time is updated with a moving average of the travel time of the hardware vehicle in the first trip. The travel time for the simulated vehicle in the second trip has been affected by the hardware
vehicle’s first travel time. In this way, the parallel reality framework updates the simulated vehicle’s estimated travel time. As shown in Figure 5.7, the actual travel time for the simulated vehicle on the link converge to the travel time of the hardware vehicle, which matches our expectations.

It is important to synchronize the simulation and the reality, because the simulation is based on events whose time of occurrence is governed by the simulation clock, while the reality is based on real-time clock. Although the clocks may be different, the causality of the system must be maintained. The synchronization and causality can be preserved by including the reality events into the Simulation Coordinator and Scheduler, so that the sequential orders of the events are enforced in the event queue maintained by the Scheduler. For example, the simulated vehicle is not supposed to start moving on the path if the real vehicle is still on the path. This means that the event for the simulated vehicle to start moving can only happen after the event for the physical vehicle having triggered the sensor to indicate that it has arrived at the destination. Even though the simulation time may have passed the scheduled time for the event that the simulated vehicle begins to move on the link, the Simulation Coordinator will need to hold this event until the it receives the event from the sensor system that the physical vehicle has arrived at the destination.

5.5 Summary

In this chapter, we have demonstrated applicability of the SPEED-CS framework for parallel reality systems. A parallel reality system consists of software models and
physical models. Physical models are data sources for parameters of software models in the simulation. The framework provides a working platform for maintaining the fidelity of a simulation adaptively. The framework is a multi-agent based architecture, in which “data exchange agents” manage the collection, dissemination, and analysis of data from dynamic data sources including simulations and/or physical systems. The messages passed among the models in the framework use XML as a common data model for information exchanged among systems. An experiment involving simulation of miniature of autonomous vehicles in a laboratory setup is developed to show the applicability of the framework.
6.1 Conclusions

The research in this dissertation involves in the design and implementation of a platform for distributed computing system, which is called the SPEED-CS platform. The goal of the platform is to provide a framework for system design as well as a test-bed to experimenting and evaluate the system. The platform is required to support distributed computing system, to be able to fusion data from different data source, and to help monitor performances of the experimental system.

Distributed simulation is widely used for research of complex and large-scale systems. There are different methods to decompose a simulation and to implement time management. In order to fully explore the parallel relationships of sub-simulations, space-parallel decomposition is applied in the SPEED-CS framework. The space-parallel decomposition allows a simulation to be decomposed into sub-models or components in state space domain. Each component is assigned to a process, while components exchange information with each other according to the coupling relations. The SPEED-CS applies DEVSJAVA as the discrete event simulation engine, which supports space decomposition of simulation models. There are two paradigms to construct a distributed simulation. One is Specific Tools Paradigm, which design a distributed simulation from scratch for a specific research
area. Many examples of this method can be found in network simulation researches. Another paradigm is the federate paradigm, which emphasizes the interoperability among simulations and reusability of simulation models. High Level Architecture (HLA) has been accepted as an IEEE standard for federate paradigm. Using DEVS specification, the SPEED-CS platform is able to provide most of the services that HLA specifies.

Distributed simulations can help speed up the calculations, however they also introduce the complexity of distributed data source over the network. One straightforward method is to have all the sub-simulation report to a centralized data source for analysis. The disadvantage of this method is that it may generate large amount of network traffic to cause the network congestion and thus slow down the simulation. In our SPEED-CS platform, we design a multi-agent system (MAS) for information collection under distributed data sources. We show that the framework can reduce the network traffic significantly.

Distributed data sources include not only data generated by sub-simulations, but also real-time data sources (such as sensor data) as well. The time management in simulations with these data sources is critical. The SPEED-CS platform applies conservative time management scheme to maintain the causality of hybrid systems that include sub-models evolving according to different time systems. A typical application of this kind of hybrid system is parallel reality, in which real-time data are collected to ensure the model fidelity in the simulation. We show the applicability of the framework through an experiment called “open-bit mine”.
Many complex problems require the integration of simulations and optimizations. For example, stochastic programming can use simulations as a scenario generator for optimization models; and in some other cases, simulations need optimization models to help configure parameters. The SPEED-CS platform is shown to be able to provide various services to help integrate simulations and optimizations. An example of production mix optimization and simulation is designed to illustrate the services of the platform.

6.2 Future Research Directions

Although the dissertation demonstrated that the SPEED-CS platform is multi-agent-based, scalable, and generic framework to integrate distributed simulations, optimizations, and real-time systems, there are still a lot of challenges remain to extend the work. The following are some of the research area along the line of work to be explored:

1. The SPEED-CS platform has designed and implemented an I/O automaton — the Data Exchange Agent. The data exchange agent is able to migrate from sub-simulations to sub-simulations. However the current version of data exchange agent relies on a pre-defined path. In order to extend the intelligence of the data exchange agent, we may allow the agent to make decisions of which sub-simulation to visit next based on conditions of the sub-simulation. This will require the data exchange agent to have a decision-making algorithm implanted.
2. Security is always an issue for distributed systems. For the SPEED-CS framework, there are many sections requiring security check for the system. For example, when a data exchange agent is going to join the sub-simulation, the sub-simulation is allowed to decide whether to include the agent. For the integration of simulations and optimizations, the accessibility of the list of the models needs to be examined too. There have been many research results on generic security issue of network systems, which can be applied in the SPEED-CS framework.

3. In the parallel reality research we have designed the “open-bit mine” experiment to demonstrate the applicability of the framework. It is worthwhile and interesting to develop more complicated hybrid systems based on the “open-bit mine” experiment. The framework can be used to evaluate different path designs of the map. We can also explore the performance of different processors of the physical vehicles. The framework allows higher-level operations control such as to design and validate different collision avoidance algorithms. Different wireless protocols between vehicles and the transceiver or wireless communication hardwires can be experimented as well.
In this simulation, the mine graph has four loading drives and four unloading drives and six trucks are involved in complex tasks. The simulation is divided into four geographical areas. As shown in Figure A.1, each quadrant is a sub-simulation. The trucks are modeled with many details: they simulate the acceptance of commands and are able to solve the contention over the links. The trucks are mobile agents and they can be transferred from one simulation to another. Figure A.1 is the initial state of the simulation.
The six trucks are denoted with dots of different colors. In the first quadrant is a gray truck. There are two trucks in the second quadrant, denoted with blue and light blue dots. The two trucks in the third quadrant are yellow and black. The fourth quadrant contains the red truck. We explain the movement of the red truck in detail since every truck has similar behaviors.

Figure A.2 Simulation snapshot: red truck to migrate.

After the simulation starts, the trucks move according to the commands they receive from the control center. The snapshot of the simulation is shown in Figure A.2. Note that the red vehicle is moving close to the border of the maps in quadrant I and IV.
In Figure A.3, the red truck has crossed the boarder and joined in the sub-simulation with the map of quadrant I. Unlike experiment 1 where the truck position is passed to the sub-simulation, the truck object is passed to the sub-simulation and thus more packets are involved in this transfer.

Figure A.3 Simulation snapshot: red truck finishes migration.

Finally the red truck approaches the destination of its first trip as shown in Figure A.4.
Figure A.4 Simulation snapshot: red truck arrives at destination.
APPENDIX B  THE GRAPHIC USER INTERFACE OF PARALLEL REALITY SYSTEM

In parallel reality system, a graphical user interface is developed to provide users with friendly interface to use coordination services. Users have to first set up the system with a Setup GUI and then are able to monitor the system with a Simulation GUI. A snapshot of the Setup GUI is shown as in Figure B.1.

A user can perform the following functions:

1. Selects the path planning algorithm:

Currently two path-planning algorithms are implemented. They are the Dijkstra’s shortest path algorithm and the congestion control algorithm. The user selects one of these to be used in the simulation.
2. Selects the server hosts:

The user can select the hosts for the virtual server and the reality server. The virtual server is where the virtual simulation is run and the reality server is the server where the programs for communicating with the real vehicle are run.

![Select server hosts](image)

**Figure B.3 Select server hosts**

The user can add or remove the servers to be used. For this the Add/Remove Servers button can be used. The coordination program reads the possible server name from the server files upon startup.

![Add/Remove a server](image)

**Figure B.4 Add/Remove a server**
3. Selects real vehicles:

The user needs to select the real vehicle to be run by the simulation. The reality server runs the particular real vehicle based on this selection. The provision for adding and removing real vehicles is provided.

![Image of real vehicle selection interface](image)

Figure B.5 Select real vehicles

4. Controls the simulation:

The SET button enables the coordination program to lookup the servers and enables it to get ready to run. The START button starts the simulation and the STOP button stops it. The PAUSE button is used to temporarily hold the simulation and the button caption will change to RESUME after it is clicked. Clicking RESUME continues the simulation.
After the system is set up, the simulation window shows up. The simulation window displays the layout of the map, and the positions and movements of the vehicles. The \textit{Simulation Coordinator} collects the virtual vehicle information directly from the virtual simulation model. The information from the real vehicle is provided by the sensor system to the software execution model which is fed back to the \textit{Simulation Coordinator}. In Figure B.7, the red dot represents the real vehicle and the black dot represents virtual vehicle.
REFERENCES


Boudoukh, T., M. Penn, and G. Weiss (2001). Scheduling job shop with some identical or similar jobs. *Journal of Scheduling* 4, 177-199.


Eker, J., C. Fong, J.W. Janneck, and J. Liu (2001). Design and simulation of heterogeneous control systems using Ptolemy II. In Proceedings of the IFAC
http://ptolemy.eecs.berkeley.edu/publications/papers/01/controlNTCC.


