A framework for component-based distributed simulation on the grid : execution support

Wang, Yong

2008

Wang, Y. (2008). A framework for component-based distributed simulation on the grid : execution support. Master' s thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/13580>

<https://doi.org/10.32657/10356/13580>

**NANYANG
TECHNOLOGICAL
UNIVERSITY**

**A FRAMEWORK FOR COMPONENT-BASED
DISTRIBUTED SIMULATION
ON THE GRID: EXECUTION SUPPORT**

WANG YONG

SCHOOL OF COMPUTER ENGINEERING

2008

A FRAMEWORK FOR COMPONENT-BASED DISTRIBUTED SIMULATION
ON THE GRID: EXECUTION SUPPORT

WANG YONG

2008

A Framework for Component-based Distributed Simulation on the Grid: Execution Support

Wang Yong

School of Computer Engineering

A thesis submitted to the Nanyang Technological University in
fulfillment of the requirement for the degree of Master of Engineering

2008

Statement of Originality

I hereby certify that the content of this thesis is the result of work done by myself and has not been submitted for a higher degree to any other University or Institution.

.....

Date

.....

Signature

Abstract

Nowadays, computer simulation, the modeling of a physical system on a computer, plays an important role in various areas such as industrial production, business, education, engineering and science. As the complexity of the simulation system keeps increasing, more and more computing resources are required to carry out the simulation. The Grid is defined as coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations. It provides services for distributed computing, such as sharing computing power and data storage capacity over the Internet. It is therefore considered to be an ideal environment to run large-scale simulation systems. However, there is still no mature framework for developing and conducting simulations on the Grid. Thus a Framework for Component-based Distributed Simulation on the Grid (SOAr-DSGrid) is proposed. It is a service-oriented, component-based and distributed framework that supports the execution of large-scale simulations.

A Service-Oriented Architecture (SOA) is essentially a collection of independent services with well-defined interfaces that can be accessed without knowledge of their implementation. The architecture is widely used to achieve resource virtualization, on-demand provisioning, and service (resource) sharing. With SOA, it is much easier for clients to invoke the services provided by the system.

A component-based approach separates the whole system into different parts based on their functionalities. The approach provides better reusability as components with well defined interfaces can be used again in other systems easily. Developers can follow the component-based approach to develop high quality systems.

Fully distributed simulation execution is the most important characteristic of the framework, and is the key for the framework to support large-scale simulations. It means that all simulation components can directly communicate with each other without in-

volving a central component. It eliminates the bottleneck problem when too many components need to communicate with a central component in a traditional framework. The distributed time manager and rule engine are two crucial features that make the simulation execution fully distributed.

The distributed time manager is in charge of logical time and guarantees that the whole simulation system is causally correct. The time management algorithm makes use of simulation topology information to reduce constraints on time advancement between simulation components. Various experiments conducted on a cluster demonstrate that a fully distributed time management system has better scalability and efficiency than a centralized version.

The rule engine enables distributed data flow and control flow within the framework. Data flow describes the way that data moves in the system while control flow determines the order of activities that happen in the system. There are two types of rules: operation rules and routing rules. Operation rules are designed to select the next entity to be processed and the appropriate operation based on available component resources. Routing rules on the other hand decide to which component the processed entity will be sent. A case study using the rule engine to implement various dispatching rules in a simulation for multi-recipes wet benches in the wafer fabrication process is conducted. The case study shows that different dispatching rules can be implemented easily to cater for varying requirements from the users and also that different rules can be interchanged conveniently as required.

Acknowledgement

This thesis is the result of three and half years of work where I have been accompanied and supported by many people. It is a pleasant aspect that I have now the opportunity to express my gratitude to all of them.

The first person I would like to thank is my direct supervisor, Dr. Stephen John Turner. He has been directing me since my final year project when I pursued my degree back in 2003. During these years, I have known that Dr. Turner is a person with enthusiasm and an integral view on research. His mission for providing high quality work has made a deep impression on me. I owe him a lot of gratitude for guiding me in these years.

I would also like to thank Professor Cai Wentong. He has cosupervised and inspired me on the project. He has given many invaluable ideas and a lot of support to my work. Without him, the project would not be this successful.

The third person to whom I would like to express my gratitude is my project partner Chen Xinjun. His initiative and enthusiasm is the key to the success of this project.

Last but not least, I would like to thank Feng Yuhong, Tang Ming, Zhu Ning, Ms Irene Goh and many other colleagues that helped me when I was in PDCC.

Contents

Abstract	i
Acknowledgement	iii
Contents	iv
List of Figures	x
List of Tables	xiv
1 Introduction	1
1.1 Simulation	1
1.2 Service Oriented Architecture (SOA) and Grid Services	3
1.3 Component-based Approach	4
1.4 Motivation of Project	5
1.5 Objectives of Project	6
1.6 Thesis Organization	7
2 Related Work	9

Master Thesis

CONTENTS

2.1	System Modeling and Simulation Models	9
2.1.1	Distributed Simulation and HLA	9
2.1.2	Time Management in Distributed Simulation	11
2.2	Grid Services	13
2.3	Workflow Execution Models	14
2.3.1	Web Services Workflow Model	16
2.3.2	Grid Services Workflow Model	16
2.4	Component-based Approach	18
2.4.1	The Common Component Architecture (CCA)	19
2.4.2	Component-Expert Architecture (CEA)	20
2.5	Distributed Simulation on the Grid	22
2.6	Summary	23
3	Preliminary Study: Grid Services for Distributed Simulation	26
3.1	Overview of the Framework	26
3.1.1	Index Service	28
3.1.2	RTI Factory Service	28
3.1.3	Federate Factory Service	29
3.2	Design Issues of Major Services	30
3.2.1	Index Service	30
3.2.2	RTI Factory Service	31
3.2.3	Federate Factory Service	34

Master Thesis	CONTENTS
3.3 Performance	38
3.4 Summary	41
4 A Framework for Component-based Distributed Simulation	43
on the Grid (SOAr-DSGrid)	
4.1 Principles of the Design	43
4.2 Concepts of the Framework	44
4.2.1 Components	45
4.2.2 Entities	45
4.2.3 Rules	46
4.2.4 Variables	46
4.3 Overview of the Framework	48
4.3.1 Service Provider	49
4.3.2 Client	49
4.3.3 Simulation Component	50
4.3.4 Organizer	51
4.3.5 Index Service	51
4.3.6 Repository	51
4.4 Design of Primitive Component	52
4.4.1 Component Manager	53
4.4.2 Operations	53
4.4.3 Entity Queues	57

Master Thesis**CONTENTS**

4.4.4	Internal Event Queue	57
4.4.5	Local Rule Engine and Rules	58
4.4.6	Shared Variable Manager	59
4.4.7	Local Time Manager	60
4.5	Executing a Simulation Session	60
4.6	Summary	62
5	Execution Support	64
5.1	Fully Distributed Simulation Execution	64
5.2	Local Time Manager	65
5.2.1	Interactions between Components	65
5.2.2	Design of the Local Time Manager	66
5.2.3	Implementation of the Local Time Manager	69
5.3	Local Rule Engine and Rules	77
5.3.1	Design of Operation Rule and Routing Rule	78
5.3.2	Design of the Local Rule Engine	83
5.3.3	Implementation of the Local Rule Engine	84
5.4	Summary	84
6	Evaluation of the Local Time Manager	86
6.1	Experiment Setup	86
6.1.1	Centralized Time Manager	86
6.1.2	Simulation Models	87

Master Thesis	CONTENTS
6.1.3 Experimental Results	90
6.2 Summary	95
7 Case Study: Use of the Local Rule Engine	97
7.1 Multi-Recipes Wet Benches in the Wafer Fabrication Process	97
7.1.1 Generator and Wet Bench Allocation Component	98
7.1.2 Batch Processing Component	104
7.2 Objectives of the Case Study	109
7.2.1 Experiment Design	110
7.2.2 Experimental Results	111
7.3 Summary	121
8 Conclusions and Future Work	123
8.1 Achievements	123
8.2 Future Work	125
Appendices	134
A Component Interface Description File	135
B Composite Component Implementation File	137
C Simulation Description File	140
D Entity Type File	143

Master Thesis

CONTENTS

E Papers Published/Submitted from this Research

145

List of Figures

1.1	HLA Runtime Infrastructure	3
2.1	Globus 4 Framework	15
2.2	Web Services Workflow Model	17
2.3	Grid Services Workflow Model	17
2.4	Component Composition in Common Component Architecture	20
2.5	Header of a CEA Component	21
3.1	Architecture Overview	27
3.2	Interface of Index Service	32
3.3	Reservation Mechanism	33
3.4	Interface of RTI Service	34
3.5	Interface of Federate Service	34
3.6	Using the Grid Services	36
3.7	Client Pseudo Code	37
3.8	Setting up a Simulation	38
3.9	Creating a Service	39

3.10	Interaction between RTI Service and Index Service	39
4.1	Concepts of the Framework	47
4.2	Simulation Framework	48
4.3	Composite Component Structure	50
4.4	Primitive Component Structure	52
4.5	Flowchart of Component Manager of an Entity Generator	54
4.6	Flowchart of Component Manager of an Entity Processor	55
4.7	Abstract Operation Class	56
4.8	Involvement of Client Program	61
4.9	Pseudo Code of Client	62
5.1	Interactions between Components	66
5.2	Relationship between Different Parts of a Component and the Distributed Time Manager	67
5.3	Entity and Entity Queue Classes	70
5.4	Event and Event Queue Classes	71
5.5	Component Class	72
5.6	Calculation of LBTS	73
5.7	Inform Time Advanced	73
5.8	Update Time Advancement of Constrained Component	74
5.9	Three NER Cases	75
5.10	NER Algorithm	77

Master Thesis**LIST OF FIGURES**

5.11 Pending and Waking	78
5.12 Rule Type File	79
5.13 FIFO Rule File	81
5.14 Shortest Queue Rule File	82
5.15 Local Rule Engine	83
5.16 Variable Binding	84
5.17 Execution of Operation and Routing Rules	85
6.1 Example of Centralized Time Service	87
6.2 Super-Ping Model	88
6.3 N-Way Connection Model	89
6.4 Average Number of NERs Granted	91
6.5 Average Number of Entities Processed	92
6.6 PDCBL Cluster	93
6.7 Average Number of NERs Granted	94
6.8 Average Number of Entities Processed	95
7.1 System Overview	98
7.2 Bath, Recipe and Batch Entity Data Structure	99
7.3 Shared Variables used in Global Dispatching Rules	100
7.4 Batch Generator	101
7.5 Earliest Start Time	102
7.6 Earliest Finish Time	103

7.7	Same Recipe	104
7.8	Batch Processing Component Operation Part 1	105
7.9	Batch Processing Component Operation Part 2	106
7.10	Batch Processing Component Operation Part 3	107
7.11	FIFO	108
7.12	Shortest Processing Time	109
7.13	Same Recipe	109
7.14	Average Waiting Time in Seconds	114
7.15	Average Cycle Time in Seconds	114
7.16	Average Waiting Time in Seconds with Modified Recipes	119
7.17	Average Cycle Time in Seconds with Modified Recipes	120

List of Tables

3.1	Testbed Infrastructure	40
3.2	Test Results	41
4.1	Future List stored in Component A	59
4.2	History List stored in Component A	60
5.1	Regulating Components Table stored in Component C	68
5.2	Constrained Components Table stored in Component C	68
5.3	Result from the FIFO Operation Rule	81
5.4	Result from the Shortest Queue Routing Rule	82
6.1	Testbed Environment	90
7.1	Baths associated with each Wet Bench	110
7.2	Possible Recipes handled by each Wet Bench	112
7.3	Original Recipes handled by each Wet Bench	113
7.4	Experimental Results in Seconds with Original Recipes	115
7.5	Average Waiting Time in Seconds with Original Recipes	115

Master Thesis

LIST OF TABLES

7.6	Average Cycle Time in Seconds with Original Recipes	116
7.7	Modified Recipes handled by each Wet Bench	117
7.8	Experimental Results in Seconds with Modified Recipes	118
7.9	Average Waiting Time in Seconds with Modified Recipes	119
7.10	Average Cycle Time in Seconds with Modified Recipes	120

Chapter 1

Introduction

1.1 Simulation

Simulation is “the process of designing a computerized model of a system (or process) and conducting experiments with this model for the purpose either of understanding the behavior of the system or of evaluating various strategies for the operation of this system” [41]. Nowadays, computer simulation, the modeling of a physical system on a computer, plays an important role in various areas of industrial production, business, education, engineering and science. It is a powerful tool for investigating and evaluating complex scenarios which are too costly or too dangerous for experimentation using a physical system.

With the evolution of networking technology, it is possible to construct large-scale distributed simulations using a huge number of geographically distributed computing resources. One of the currently available technologies to build a large-scale distributed simulation is the Runtime Infrastructure provided by the High Level Architecture. The High Level Architecture (HLA) is an IEEE standard for modeling and simulation, aimed to facilitate interoperability among simulators and reuse of simulation components [10,32]. In addition, HLA enables computer simulators to communicate with other computer

simulators regardless of the computing platforms. The HLA consists of the following four parts [1]:

- HLA Rules.

These are the rules that the simulation and simulators must obey to be compliant to the standard.

- Interface Specification.

The interface specification defines how HLA compliant simulators interact with the Runtime Infrastructure (RTI). The RTI provides a programming library and an application programming interface (API) compliant to the interface specification.

- Object Model Template (OMT).

The OMT specifies the format of information that is communicated between simulators and how it is documented.

- Federation Development and Execution Process (FEDEP) Model.

The purpose of FEDEP is to describe a generalized process for building HLA federations.

In HLA terminology, an HLA compliant simulator is referred to as a federate. Multiple simulators connected via the RTI using a common Federation Object Model (FOM) defined according to the OMT are referred to as a federation. A collection of related data sent between simulators is referred to as an object. Objects have attributes (data fields). Events sent between simulators are referred to as interactions. Interactions have parameters (data fields). Figure 1.1 illustrates the architecture.

Figure 1.1: HLA Runtime Infrastructure

1.2 Service Oriented Architecture (SOA) and Grid Services

A Service-Oriented Architecture is essentially a collection of services. It is an evolution of distributed computing based on the request and reply design paradigm for synchronous and asynchronous applications [29]. The loosely coupled nature is the key to these services, i.e. the interface is independent of the implementation. Different services can be integrated together to build a new application without knowing their underlying implementations. A Service-Oriented Architecture has the following four key characteristics:

- SOA services have self-describing interfaces in platform-independent XML documents. Web Services Description Language (WSDL) [8] is the standard used to describe the services.
- SOA services communicate with messages formally defined via XML Schema (also called XSD) [45]. Communication between consumers and services typically happens in heterogeneous environments, with little or no knowledge about the services. Messages between services can be viewed as key business documents processed in an enterprise.

- SOA services are maintained in the enterprise by a registry that acts as a directory listing. Applications can look up the services in the registry and invoke the service. Universal Description, Definition, and Integration (UDDI) [39] is the standard used for a service registry.
- Each SOA service has a quality of service (QoS) associated with it. Some of the key QoS elements are security requirements, such as authentication and authorization, reliable messaging, and policies regarding who can invoke services.

In recent years, Grid services have gained more and more attention. They are an extension of existing Web services, with added features, such as lifecycle management, service data, and notifications. The Grid is distinguished from conventional distributed computing by its focus on large-scale resource sharing, innovative applications, and, in some cases, high-performance orientation [18]. The Grid concept is defined in [19] as coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations. The essence of the Grid is the federating of computing resources to accelerate application processing, plus the virtualization of these resources. At its core, the Grid is all about distributed computing and resource management. The Open Grid Services Architecture (OGSA) was developed to define an open standard for interaction and encourage interoperability between components supplied from different sources [27]. The Globus Toolkit 3 is the first major implementation of the OGSA standard and Globus Toolkit 4 is the latest implementation [23].

1.3 Component-based Approach

A component-based approach normally separates the whole system into different parts according to their functionalities. For example, in E-business a three-tier component-based architecture might be designed so that the user interface is separated from the

business logic and the data storage layer [15]. With this type of architecture, it is easy to reuse certain functionalities within each layer in building a new system or for maintenance. Thus a component-based application is a newer type of application with better reusability. It is easier with a component-based application to update applications without having to rewrite them entirely. Also component based architectures, unlike legacy applications, do not have to run on a particular platform, as most of them are platform neutral.

1.4 Motivation of Project

Although HLA is popular in constructing distributed simulation systems, there are some issues associated with HLA-based simulations. In the DMSO RTI [12], the control process RTIExec is discovered by explicitly specifying the endpoint in the configuration file. This static technique lacks flexibility, which means that federates have to be mapped to computing resources before the simulation starts, and dynamic allocation is not supported by the HLA. The HLA works well when the simulation components do not need to dynamically locate their partners and all the simulation components are set up within one local network. However, nowadays more and more simulation systems are adopting large-scale distributed architectures, in which the simulation components are normally located physically apart and these components do not know the address of others in advance.

One of the advantages of such an architecture is that the cost of developing the simulation will be reduced as it is possible for clients to choose suitable components from worldwide registries instead of developing new components. In addition, with more components available, the time of developing a new simulation system will also decrease. However, as the HLA is restricted by the use of static components, it does not provide facilities to support such an architecture. Grid technology, on the other hand, provides us with resource discovery, dynamic resource locating and some other services, which

means that it is a suitable platform for this architecture. This project aims to develop an architecture based on the Grid which supports large-scale distributed simulation systems.

1.5 Objectives of Project

There are many special frameworks for the Grid to meet different demands from users, like the NERC Data Grid [34], which focuses on data discovery, delivery and use, and OODT Data Grid Framework [33], which provides Data Grid features for the automatic discovery and retrieval of resources. However, a framework which fully supports distributed simulation on the Grid has not yet emerged. Thus the general aim of this research project is to design a framework that supports component-based distributed simulation on the Grid. The framework should meet the following criteria:

- It is a Service-Oriented Architecture (SOA). The interface of the service is separated from the implementation of the service.
- It is component-based. A component-based system provides better reusability and flexibility.
- It is distributed. The execution of the simulation components is fully distributed.

After analyzing the design of the framework, two critical parts of *execution support* are identified, i.e. time management and the rule engine. The time manager manages the logical time of the whole system. If it is not well designed, it will affect the performance of the whole system. Thus in this project, the time manager is one focus of research. The specific objectives for the time manager are:

- It must ensure temporal causality among simulated events. In other words, the time manager guarantees that events happen in the correct order.

- It must be fully distributed so that it meets the distributed criteria for the framework.
- It must be easy to understand and use by simulation developers.

The rule engine is another focus of research. The specific objectives for the rule engine are:

- It must enable both data flow and control flow to be distributed.
- Rules must be easy to develop and maintain by simulation developers.

Before building the framework, a preliminary study is first conducted to become familiar with the knowledge of simulation systems and the Grid. The preliminary study includes building an experimental framework. The objective of this framework is to investigate different functionalities provided by the Grid, e.g. remote service discovery and invocation. It will enable current HLA-based simulation components to be invoked by the Grid client.

1.6 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 summarizes relevant research on system modelling and simulation, Grid services, workflow execution models, the component-based approach and distributed simulation on the Grid. The detailed description of an experimental framework that enables HLA-based Grid simulation is presented in Chapter 3. Chapter 4 discusses the design of a framework for component-based distributed simulation on the Grid. In Chapter 5, the design and implementation of the time manager and rule engine for the framework are described. Performance results for the time manager are presented in Chapter 6. Chapter 7 discusses a case study illustrating

the use of the rule engine. Finally Chapter 8 concludes the whole thesis and discusses some future work.

Chapter 2

Related Work

2.1 System Modeling and Simulation Models

2.1.1 Distributed Simulation and HLA

Computer simulation is one of the most useful and powerful of mathematical-modelling approaches in many areas, like manufacturing, distribution and logistics, health care and military applications. Distributed computing is a science which solves a large problem by giving small parts of the problem to many computers to solve and then combining the solutions for the parts into a solution for the problem [40]. Recently the need to simulate more and more complex systems means that computer simulations can require large amounts of computing power. Thus distributed simulation, which involves conducting computer simulations in a distributed manner, emerges. There are two advantages of running simulations in a distributed manner.

- Openness:

A well-designed distributed simulation system normally has open standards, e.g. HLA. Thus, different researchers can easily form a new system by combining existing components or extending an existing system.

- Scalability:

The distributed simulation system can achieve better scalability than a centralized simulation system. With the increasing size of simulation system, scalability becomes a critical factor that affects the performance of a simulation system.

However, there are also some drawbacks for distributed simulation. If not planned properly, a distributed simulation system can decrease the overall reliability of computation. The flexibility provided by a distributed simulation also brings a higher initial development cost if there is no standard procedure to build a simulation system. For this reason the High Level Architecture (HLA), which is a general purpose architecture for distributed computer simulation, was developed. As introduced in the last chapter, HLA is founded on four major components, i.e. HLA Rules, Interface Specification, Object Model Template (OMT) and Federation Development and Execution Process (FEDEP) model.

The HLA rules describe the responsibilities of the overall federation and the federates that join the federation.

The interface specification can be further categorized into the following service groups.

- Federation Management
- Declaration Management
- Object Management
- Ownership Management
- Time Management
- Data Distribution Management
- Management Support Services

The Object Model Template (OMT) provides a common format for communication between HLA simulators. The OMT is used to define the structure of the following documents:

- Federation Object Model (FOM). The FOM describes the shared objects, attributes and interactions for the whole federation.
- Simulation Object Model (SOM). A SOM describes the shared objects, attributes and interactions for a single federate.

The intent of the FEDEP model is to specify a set of guidelines for federation development and execution that federation developers can leverage to achieve the needs of their application [43]. There are six steps in total defined in the FEDEP model.

- Define federation objectives.
- Develop federation conceptual model.
- Design federation.
- Develop federation.
- Integrate and test federation.
- Execute federation and prepare results.

2.1.2 Time Management in Distributed Simulation

Time management is a critical part of a simulation system because of causality and repeatability. The purpose of a time management scheme is to ensure temporal causality among simulated events. The design of time management is relatively easy in a single computer simulation as all the events occur in a logical sequence corresponding to the real

events being simulated. However, it becomes more complex in a distributed simulation as multiple components can possibly advance their time at a different pace. The essential problem in the design of time management is to adhere to the local causality constraint. The local causality constraint means that events within each simulation component must be processed in timestamp order. Distributed simulation algorithms can be broadly classified into conservative and optimistic based on how they enforce the local causality constraint [21]. A conservative algorithm achieves this by not allowing an event to be processed by a simulation component until it can ensure that the simulation component will not subsequently receive any other event that happened earlier. Optimistic time warp allows violations of local causality to occur, but will detect them at runtime and recover using a rollback mechanism [28].

In HLA, in order to achieve its interoperability requirements, time management must appear transparent to all federates in a given federation [7]. Any federate can set or unset two switches, time-constrained and time-regulating. Events are delivered to time-constrained federates in timestamp order. Events are delivered to non time-constrained federates in the order that they are received by the federate. Federates that are time-regulating schedule events that are delivered in timestamp order. Non time-regulating federates schedule events that are delivered in the order received by the destination. The HLA RTI provides the following services related to time management:

- Transportation Services

The categories of transportation service are distinguished by (i) reliability of event delivery, and (ii) event ordering. There are two levels of services provided by the RTI. Reliable delivery guarantees that any message sent using this service will arrive intact at the proper destination and best effort delivery reduces latency, but does not guarantee delivery. Event ordering characteristics specify the order and time at which events may be delivered to the federates and are central to the

HLA time management services. There are two ordering mechanisms. Receive order (RO) events are delivered to the federate according to the sequence that they are received. Time Stamp Order (TSO) events are delivered to time constrained federates in timestamp order.

- Time Advance Services

Time advance services provide a protocol for the federate and RTI to jointly control the advancement of logical time. The RTI can only advance a time constrained federate's logical time to T when it can guarantee that all TSO events with timestamp less than or equal to T have been delivered to the federate ^{2.1}.

2.2 Grid Services

In recent years, Grid technology has gained more and more attention from the commercial and scientific areas because of its facilities for resource coordination, open-standards and high efficiency. The Open Grid Services Architecture (OGSA) [20] aims to standardize Grid services. The OGSA represents an evolution towards a Grid system architecture based on Web services concepts and technologies.

The Globus Toolkit was developed to allow users to program Grid-based applications [23]. In June 2003, Globus Toolkit 3 was released. This version contains an open source implementation of the Open Grid Services Infrastructure (OGSI) [47], which describes the standard interfaces and behaviors of a Grid service, and provides a technical specification of the OGSA. Globus Toolkit version 3 (GT3) also includes Globus XIO Framework and Drivers, which is an extensible I/O library, GridFTP, which provides normal FTP functionalities and also improved firewall support, and GRAM, the Grid Resource Allocation and Management. Our preliminary study is carried out using Globus

^{2.1}Depending on the time advance service used, the guarantee may be that all TSO events with timestamp strictly less than T have been delivered.

Toolkit 3.2. In GT3, Grid services are instantiated from service factories.

Globus Toolkit version 4 (GT4), released in April 2005, represents a significant advance over earlier GT versions in terms of software quality, functionality, and standards conformance [24]. The most important difference from GT3 is that the Web Services Resource Framework (WSRF) [9] replaces OGSF to specify the stateful services which are required by OGSA. WSRF now becomes the infrastructure on which OGSA is built. By adopting WSRF, there is no need to instantiate Grid services from a service factory every time, instead only a service resource object needs to be created in order to keep the state of this service request session. In addition, GT4 includes not only services which are provided in previous versions but also new services like Workspace Management service (for dynamic accounts), Data Replication Service, Grid Teleoperation Control Protocol service, and additional Monitoring and Discovery services. In addition, GT4 provides probably the most fully-featured implementation of Web services security available anywhere. Figure 2.1 taken from the Globus website [44] shows the constituents of GT4.

2.3 Workflow Execution Models

A workflow language plays an important role in service composition in a Service-Oriented Architecture. It can be used to combine several services to form a larger system. The workflow not only defines how services are composed but also describes how the tasks will be executed by this composite service. Two existing workflow models are introduced in the following subsections. They differ in the way they manage control flow and data flow. Control flow determines the order of activities that happen in the model and data flow describes how data moves in the model.

Figure 2.1: Globus 4 Framework

2.3.1 Web Services Workflow Model

Web Services Flow Language (WSFL), originally proposed by IBM, provides a means to formally specify business processes and interaction protocols [35]. In 2003, Business Process Execution Language (BPEL) was proposed to supersede WSFL by BEA Systems, IBM, Microsoft, SAP and Siebel Systems [26]. In both WSFL and BPEL, the central engine plays a key role in the whole model. It requires that all data flow and control flow pass through or originate from the engine so that the engine is able to make a decision for each step of the execution. This simplifies the design; however, it also makes the engine become the bottleneck of the whole system. While in most cases, the data flow will closely follow the control flow, it is quite possible that the way that information flows through the process is different than the sequence of activities that are invoked [42]. Figure 2.2 shows this design.

2.3.2 Grid Services Workflow Model

The Grid Services Flow Language (GSFL) is an effort to examine technologies that address workflow for Web services and leverage this technology for Grid services [31]. This project contains both an XML schema definition [48] for the specification of workflow in Grid environments and a reference implementation of the Workflow engine for use in the Open Grid Services Infrastructure (OGSI). The most significant difference between GSFL and WSFL is that GSFL allows each service to deliver data to another service without routing to the central engine. The central engine now is only responsible for the control and some small amount of data flow. Figure 2.3 shows this design.

Figure 2.2: Web Services Workflow Model

Figure 2.3: Grid Services Workflow Model

This design theoretically solves the bottleneck problem when there is too much data transfer among the system services as now only control flow needs to go through the central engine. It eliminates unnecessary data flow between the central engine and individual services. However the Grid Service Workflow Model still requires a central engine that decides the direction of data flows. This may still potentially restrict the scalability and availability of the whole system. For example, in a large-scale system, there may be hundreds of thousands of different components sending data to each other. If all the control data are sent from a central engine, the bottleneck problem may still appear. In addition, if the central engine goes down, the whole system may have to stop or at least roll back to an earlier stage which increases the instability of the system.

2.4 Component-based Approach

The scale of today's information systems has increased rapidly because of the increasing needs of the user and support from more powerful hardware and software. The normal practice of developing a software system in one cycle by a single team becomes more and more impractical. In addition, today's systems are continually evolving to cater for the needs of the client. These new phenomena require a new approach to design the system. One of these new approaches is the component-based approach. In this approach, the system is built from many components, in which the components are versionable, programmable and efficient. This makes software easier to write and reuse so that a wide choice in services, tools, languages and applications are provided [2, 25, 46].

A component-based approach is also widely adopted in constructing simulation systems. Components appear to be the ideal level of granularity for implementing simulation models [5]. Through reusing validated simulation components we will be able to reduce the costs of development and validation of simulation models [16]. Composability is the capability to select and assemble reusable simulation components in various combina-

tions into simulation systems to meet user requirements [50]. There are two main types of composability, syntactic and semantic. Syntactic composability is concerned with the compatibility of connection details, i.e whether a set of components can be combined, and semantic composability is concerned with the validity of the composition, i.e. whether this combination is meaningful. There is some research on syntactic and semantic composability [4, 11, 49, 50].

In the following subsections, two component-based architectures will be introduced.

2.4.1 The Common Component Architecture (CCA)

The Common Component Architecture [17] is a component architecture defining standards necessary for the interoperation of components developed in the context of different frameworks. It is a common architecture for building large scale scientific applications. CCA-based components are compliant with Grid standards. CCA components are connected by their ports. A *provides* port represents a functionality a component provides to other components. A *uses* port represents a functionality a component may use; the functionality is provided by a provides port of another component. Figure 2.4 shows this model.

There are several implementations of the CCA currently [30]. One of them is XCAT3. XCAT3 provides a model for building Grid applications by composing components that are located on remote resources on the Grid [52]. The XCAT3 framework allows for CCA components to be compatible with the OGSI specification. XCAT3 can be used to create a distributed application consisting of components and OGSI services.

Figure 2.4: Component Composition in Common Component Architecture

2.4.2 Component-Expert Architecture (CEA)

The Component-Expert Architecture (CEA) is an architecture based on the popular component approach and makes use of a rule-based expert system for portlets selection [14]. Portlets are pluggable user interface components that are managed and displayed in a web portal. A CEA component consists of the header and the component code. The header, which has a component type (TID) and an attributes list, describes the general purpose of the component and its specialization (see figure 2.5). As the architecture name suggests, the expert system is able to select the best component for a given unambiguous context among several candidate components. The call-environment (ENV) presents the context of the invocation and it facilitates proper selection of components. With the type of the searched component and the call-environment, the expert system will try to find a component with a specialization that is most suitable for the call-environment. The expert system will apply rules to find the most suitable component. These are expressed in a typical form, i.e., if “log-expr” then “action” else “action”. The rules are provided

Figure 2.5: Header of a CEA Component

by programmers at the development stage. If with the given rules, the expert system is still unable to make a decision, then it will take an arbitrary general component. The normal execution procedure of the CEA system is as follows [13].

- The system requires a service for the TID. It requests from the Expert Component Management Subsystem a handle to the particular component most suitable for ENV.
- The Expert Component Management Subsystem parses information obtained from the Component Container and uses the Expert System to search for suitable components for ENV.
- The Expert System gets information about all TID type components from the Components Container.
- Based on the rule-based knowledge of the Expert System and on the External Knowledge Base, as well as on the specialization of components of the required type, the Expert System makes a decision as to which component is most suitable for ENV.

- If the Expert System finds the appropriate component, then the Expert Component Management Subsystem returns a handle to the component, otherwise an error message is returned.
- The program executes the component referenced by the handle.

2.5 Distributed Simulation on the Grid

Managing resources for distributed simulation has been attracting researchers' attention for many years. In [36], a resource sharing system which aims to dynamically map federates to compute resource in order to achieve load balancing is proposed. The resource management issue is also addressed in [6] which proposes a load management system for HLA-based simulation on a Grid environment.

Recently some research effort has focused on combining Web or Grid technology and HLA in order to take advantage of both for distributed simulations. There have been various projects designed to conduct distributed simulation on the Grid. An interesting approach to combining web technology and the HLA is presented in [37], where RTI calls are formatted via SOAP and a BEEP communication layer is employed to enable bi-directional calls and call backs via the RTI. A system supporting execution of HLA distributed interactive simulations in a Grid environment is presented in [53], where federate migration is supported by the Grid management system. The key difference of that approach from the proposed framework in this paper is that users have to provide HLA simulation code in the former, while they can make use of the simulation models discovered in the latter to compose a large-scale simulation. In [54], the idea of a three-level approach to building the Grid services for HLA-based applications is described.

There are also some efforts to develop frameworks for building simulations on the Grid directly. In [55], an approach is presented to build distributed simulations using Grid services based on GT3 Core, realizing the separation of simulation resources and simulation applications with a server responsible for the organization of simulation resources. In that system, there is no standard high level communication protocol defined, i.e. the same set of data may be transferred in different ways in different simulation models. This restricts the reusability of existing models as different developers tend to use different communication mechanisms. In [56], a Grid-based Distributed Simulation Architecture (GDSA) is proposed to solve four pending problems in distributed simulation: scalability, communications, management mechanism and QoS insurance computing environment. The architecture provides resource management and scheduling and is claimed to support both HLA federates and normal Grid simulation applications. It allows HLA-based models to be run in the architecture by wrapping HLA RTI services into Grid services. However it does not provide a solution on how HLA federates can communicate with normal non-HLA simulation components in the same simulation session. This constrains the reusability of the existing simulation components.

2.6 Summary

In this chapter, we describe research in the areas that are related to the design and development of our simulation framework on the Grid. The research areas include system modeling and simulation models, Grid services, workflow execution models, component-based approaches and distributed simulation on the Grid.

For system modeling and simulation models, we discuss the High Level Architecture (HLA) and the design of a time manager in a distributed simulation system. The HLA is a general purpose architecture for distributed computer simulation and is widely used. However, many implementations of the HLA use a centralized time manager to manage

logical time, and this may lead to poor performance when the scale of the system increases. Time management is a critical part of a distributed simulation system. There are two fundamental approaches, conservative and optimistic, which differ in how they enforce the local causality constraint. The conservative approach advances logical time when it can ensure that no past events will be received while the optimistic approach allows a violation of the local causality constraint and recovers with a rollback. The conservative approach is chosen for our framework as the approach is simpler in a distributed environment.

The Grid is designed for running large-scale distributed systems. The latest Globus Toolkit version, GT4, implements the web services resource framework (WSRF) and it provides better resource management, resource sharing and service location than the previous version. We decided to use GT4 to implement the simulation components in our framework, however we used GT3 in our preliminary study as this study was conducted before GT4 was published.

Two types of workflow execution models, WSFL and GSFL, are discussed in this chapter. These two models differ in how they implement the data flow and control flow. Data flow describes the way that data moves in the system while control flow determines the order of activities that happen in the system. WSFL adopts a central data flow and a central control flow while GSFL adopts a distributed data flow and a central control flow. After analyzing the two workflow models, we conclude that only when both data flow and control flow are distributed, the system can achieve the best scalability. Thus in the framework, we decided to make both data flow and control flow distributed.

A component-based approach provides better reusability. It can reduce the cost of creating new simulation systems. The framework adopts a component-based approach to the design of simulation of components so that simulation developers can reduce development cost and time.

Combining the above technologies, this thesis proposes a distributed, service-oriented

and component-based framework for distributed simulation on the Grid.

In the next chapter, an approach that allows HLA simulations to be conducted in a GT3 environment is discussed. This is considered a preliminary study to the whole project. It provides a better understanding of the constraints of current approaches to conducting distributed simulations on the Grid. It also inspires the design of a dedicated framework to support simulation in a Grid environment which is introduced in Chapter 4. In the thesis, we introduce a fully distributed time manager and a rule engine that enables both distributed data and control flow.

Chapter 3

Preliminary Study: Grid Services for Distributed Simulation

3.1 Overview of the Framework

In this preliminary study, a framework for executing large-scale HLA-based distributed simulations over the Grid is designed and developed ^{3.1}. It is illustrated in Figure 3.1. It consists of several key Grid services, namely the index service, the RTI factory service and the federate factory service, which are all implemented based on Globus Toolkit 3.02. The index service acts as an information server, facilitating dynamic discovery of the RTI service and federate factory service. There can be different types of federate factory services, possibly implemented by different organizations with each producing federate services for a certain type of federate.

^{3.1}This preliminary study was conducted with project officer Zhong Wenbo from PDCC, Nanyang Technological University. The index service and RTI factory service were developed by Zhong Wenbo and the federate factory service was developed by the author.

Figure 3.1: Architecture Overview

3.1.1 Index Service

The main function of the index service is to keep a record of the mapping between federations and handles of RTI service instances, support queries from federate clients, and support state updates by the RTI service. Hence, it provides a set of operations to the clients, a set of operations to the RTI service and another set to the federate factory service. With the index service, different models can be located dynamically. It is a well known service so that clients can resolve to it directly.

3.1.2 RTI Factory Service

The RTI factory service creates RTI service instances. The RTI service instance is created either by the network administrator that manages the computing services, or upon a request from clients as needed through the RTI factory service handle obtained from the index service. Each RTI service instance monitors the execution of one RTIExec process, the daemon that manages the execution of simulation federations. The RTIExec will be invoked upon request and destroyed when not used for a certain period of time by the RTI service in order to save system resources. When an RTI service instance is created, it registers itself with the index service; and similarly, when it is destroyed, it deregisters itself with the index service. The RTI service is also responsible for gathering information about the RTIExec through a monitoring thread, including the number of federations and their names, and registers and deregisters federations with the index service accordingly. As a general rule, whoever creates an RTI service instance is responsible for destroying it.

3.1.3 Federate Factory Service

A federate factory service creates federate service instances. The federate service instance can accept requests like creating a federate process, returning federate process running status or terminating a federate process from a client. Each federate service is intimately associated with a type of federate resource, e.g. a federate which simulates a racing car. This means that the federate service shall have sufficient knowledge of how to create, monitor and terminate a certain federate process. Technically this Grid service provides a new method for a client to control the federate processes that are running on remote servers. In addition, it also makes the discovery of a federate service possible if the federate factory service pre-registers itself in a well-known index service. Moreover, if a federate service acts as a client to another Grid federate service, it can also create a federate process provided by that service. This means that a federate service is able to use other federate services to form a large-scale distributed simulation system through the Grid network without the interference of a real client. In a later section, the definition of interface methods will be presented.

These services are designed such that they only provide primitive operations in their interface. In this way, high level algorithms can be implemented using the primitive operations, decoupling the services from any specific application. A typical simulation can be carried out using the framework as follows (refer to Figure 3.1). The client queries the index service to get the handle of an RTI service for the federation in which it is interested (step 1). It then contacts the RTI service to retrieve the multicast endpoint used by the RTIExec (step 2). It then queries the index service again for the handle of the federate factory service (step 3) and requests federate service instance(s) be created (step 4). The client then proceeds to instruct the federate service to create simulation federates (step 5) for that federation with the multicast endpoint passed through as a parameter. The created federates proceed to connect to the RTIExec process (step 6)

identified by the multicast endpoint and carry out the actual simulation. Depending on the actual simulation, the clients could also leave the task of querying the index service for the RTI service and retrieving the multicast endpoint to the federate services which are also able to accomplish this on their own (step 5.a & 5.b).

3.2 Design Issues of Major Services

In this section, we will look at the three Grid services introduced in the previous section in detail, examining the design and implementation issues associated with them.

In an open computing environment, clients use the framework to carry out different distributed simulations at the same time. This leads to one potential problem that clients intending to conduct different simulations may use the same federation name. In such a situation, the simulations could go wrong because federates meant for different federations end up joining the same federation. The reason is because the index service assumes the federation name is unique. As a solution to this problem, we propose a naming scheme under which federations are named with a prefix chosen by their user. The combination of the prefix and the normal federation name forms a unique name seen by the framework. The prefix can be as meaningful as a domain name, or it can be anything that makes the federation unique. With this scheme, the clients need to form logical “domains” before they actually run simulations.

3.2.1 Index Service

The index service is the central information server. All other services will register their information with the index service upon creation and update as needed. Figure 3.2 shows its interface. However, there is one consideration that must be taken into account. Assume the RTI service for some federation, say wargame, is queried, but that federation

has not been created yet. Because the RTI service only registers a federation after it has detected it, the index service has no information about federation wargame at the moment. In such a situation, one possible solution is to return an arbitrary RTI service handle, preferably the one with the lightest load (i.e. the RTI service with the least number of federations). This solution is simple but could cause problems. Imagine a situation where wargame federates A and B query for the RTI service handle almost concurrently. The request from A is processed first and handle `rtiX` which has the lightest load is returned. When the request from B is processed, the load situation may have changed. The index service may return a different handle, say `rtiY`, to B. Should this happen, federates intending to join the same federation may end up communicating with different RTIExec processes, and the simulation result would not be correct.

To solve this problem, we use a reservation mechanism that stores the returned handle in the database and returns the same handle for later queries for the same federation. The flow chart in Figure 3.3 shows how the reservation mechanism works (fed is the federation name being queried).

3.2.2 RTI Factory Service

The sole purpose of the RTI factory service is to create RTI services that do the real work. The RTI service does not function as the RTIExec process; rather it invokes the RTIExec process and monitors the execution throughout its lifetime. The RTI service either does not have an RTIExec process running, or it has exactly one RTIExec running at a time. Different RTI services will have separate RTIExec processes running with different endpoints whose IP address and port number are both generated randomly. The random generation of endpoint ensures the chance of having a clashing endpoint is almost zero. The RTI service has a separate thread that monitors the RTIExec process: when it finds a new federation running on this RTIExec, it registers the federation with the index

```

void registerRtiFactory(String facHandle);

void deregisterRtiFactory();

String getRtiFacHandle();

String getRtiServiceHandleForFed(String fed);

void registerRtiService(String handle);

void deregisterRtiService(String handle);

void registerFedWithRtiService(String fed, String handle);

void deregisterFedWithRtiService(String fed, String handle);

void registerFedFactory(String facName, String handle, String ver);

void deregisterFedFactory(String facName, String ver);

String getFedFacHandle(String facName, String ver);

```

Figure 3.2: Interface of Index Service

service; and similarly when it detects a federation has been destroyed, it deregisters the federation with the index service. The thread also accounts the idle time of the RTIExec, and terminates it after a predefined period of idle time to save system resources. The RTI service supports a query for the RTIExec endpoint and a request for creating an RTIExec process. The interface provided by the RTI service is shown in Figure 3.4.

Note that `createRti` and `createRtiWithEP` are essentially overloaded versions of the same operation; they are given different names because we want to emphasize the different usage of the two operations. The former one without any argument simply requests the RTI service to create an RTIExec process with whatever endpoint it desires, and the client can call `getMulticastEndpoint` to retrieve it. On the other hand, the latter operation requests the RTI service to create an RTIExec process with a specified endpoint and then return a flag indicating whether the process is successfully created with the requested endpoint. The differentiation of the two operations allows for flexible use under different scenarios.

Figure 3.3: Reservation Mechanism

```

boolean isRtiRunning();

String getMulticastEndpoint();

String createRti();

boolean createRtiWithEP(String endpoint);

```

Figure 3.4: Interface of RTI Service

3.2.3 Federate Factory Service

As explained in the last section, a federate factory service creates federate service instances. Through the federate service, a client can initiate a federate process, which will be managed by this federate service. After that, the client can communicate with the federate service to remotely control the running federate. The federate service contains four compulsory methods that can be accessed by clients, i.e. createFederate, createFederateWithRti, getFederateStatus and terminateFederate. Figure 3.5 lists these four interface methods, which are abstracted from their service definition file.

```

boolean createFederate(string federationName,
                      string federateName,
                      string[] federateParameters);

boolean createFederateWithRti(string federationName,
                             string federateName,
                             string rtiEndpoint
                             string[] federateParameters);

string getFederateStatus(string federationName, string
                        federateName);

boolean terminateFederate(string federationName, string
                        federateName);

```

Figure 3.5: Interface of Federate Service

The createFederate and createFederateWithRti interface methods are designed to create a federate process. They both accept the federation name, federate name and parameters of the federate as input parameters. The difference between them is that createFeder-

ateWithRti requires the multicast endpoint of an available RTIExec while createFederate does not. If createFederate is called by the client, the federate service will contact the index service and RTI service to get the multicast endpoint of an available RTIExec. In conclusion, createFederate makes the client's code simpler and createFederateWithRti gives the user flexibility to choose an RTIExec which may give better performance compared to an RTIExec arbitrarily selected by the index service and RTI service.

Before these two functions initiate a federate process, the input parameters for the federate process will be validated. This includes checking the number of the parameters and the data type. After that, a new RTI.rid file will be generated, which contains the RTIExec multicast endpoint for the federate to connect to an RTI service because a federate depends on the RTI.rid file to locate the RTIExec process. The next step is to spawn a federate process and initiate it with the validated input parameters from the client.

After a federate process is successfully created, some relevant information will be returned to the client indicating the current status of the federate process. The interface method getFederateStatus is used to acquire the current status of a running federate process. If a client finds that it has been waiting for a very long time and has still not got the result from the server, it can issue this request to check the status of the federate process. The input parameters of this method are federation name and federate name and the return value will be a string indicating the current federate process status. The last method terminateFederate is used to terminate a federate process. If a client feels that the federate process is in an abnormal state, it can issue this request to terminate the process. The input parameters of this method are also the federation name and federate name and the return value will indicate if this operation is successful or not. The above discussion reveals the general objectives of each interface method. The detailed implementation of each method shall vary according to the different types of federates that the federate

services are going to manage.

Figure 3.6: Using the Grid Services

Figure 3.6 demonstrates the use of various Grid services in the framework. There are several services in this framework diagram including the index service, the RTI factory service / RTI service and two federate factory services / federate services^{3.2}. Initially, a client does not know the address of a Grid service, it can obtain the address through querying the index service using keywords. The keywords are normally the name of the federate factory service or a short description depending on how federate factory services register themselves. After retrieving the address, the client will contact an RTI service to ask for the multicast endpoint of an available RTIExec. In the next step, the client can create one or more federate processes by sending requests to the federate service with the RTIExec multicast endpoint. The client is also able to create more federate processes by repeating this request to the same federate service or to another federate service. In this

^{3.2}For simplicity, factory services and their service instances are shown as a single box in this diagram.

framework, one federate service usually manages one type of federate process, however it can instantiate more than one instance. The different instances are distinguished by the federation name and federate name. After the instance of a federate process is created, it will join the specific RTIExec according to the multicast endpoint. The federate service is responsible for monitoring the running status of the federate process until it becomes defunct.

```

rtiHandle = index.getRtiServiceHandleForFed(federation);
while(rtiHandle == null && numberOfAttempts < limit) {

    rtiFacHandle = index.getRtiFacHandle();

    rtiFac <= resolve Rti Factory Service with rtifacHandle;

    rtiFac.createService();

    rtiHandle = index.getRtiServiceHandleForFed(federationName);

    numberOfAttempts ++;
}

rti <= resolve Rti Service with rtiHandle;

endpoint = rti.createRti();

fedFacHandle = index.getFedFacHandle(facName, ver);

fedFac = resolve Federate Factory Service with fedFacHandle;

fedLocator = fedFac.createService();

fed <= resolve Federate Service with fedLocator;

fed.createFederateWithRti(federationName, federateName,
endpoint...);

```

Figure 3.7: Client Pseudo Code

A simple client pseudo code is illustrated in Figure 3.7. Note that when querying for an RTI service, it is possible that a null string will be returned if no RTI service is running. The client can deal with this situation by proceeding to ask for the RTI factory service handle and then itself requesting an RTI service instance be created. Figure 3.8 shows the calling sequence when there is an available RTI service, and Figure 3.9 shows how to

cope with the situation when no RTI service is available. With all necessary information gathered and resources prepared, the simulation can proceed to execute. Figure 3.10 illustrates the interaction between the RTI service and index service when a simulation is running. Note that upon invocation of `registerFedWithRtiService`, the index service checks the reservation table and removes the corresponding entry if found, in order to keep the stored information consistent (refer to Figure 3.3).

Figure 3.8: Setting up a Simulation

3.3 Performance

The main objective of testing is to measure the time for setting up a simulation using the Grid services provided. The experiment is carried out on a testbed infrastructure as shown in Table 3.1. The testbed workstations are interconnected by a 100Mbps LAN.

Figure 3.9: Creating a Service

Figure 3.10: Interaction between RTI Service and Index Service

Table 3.1: Testbed Infrastructure

Process	Processor	Memory	OS
index service	Pentium III 730 MHz	512MB RAM	Redhat Linux 7.3
RTI (Factory) Services	Pentium III 730 MHz	512MB RAM	Redhat Linux 7.3
Federate (Factory) Services	Pentium III 730 MHz	512MB RAM	Redhat Linux 7.3
Test Client	Pentium III 730 MHz	512MB RAM	Redhat Linux 7.3

Six test cases are conducted: (A) time to query the index service for the RTI service handle; (B1) time to get the RTIExec multicast endpoint when the RTI service has RTIExec running; (B2) time to get the RTIExec multicast endpoint when the RTI service has no RTIExec running, including time for the RTI service to invoke the RTIExec and time to retrieve the multicast endpoint from the RTI service; (B3) time to get the RTIExec multicast endpoint when there is no RTI service, including time to query the index service for the RTI factory service handle, time to connect to the RTI factory service and create an RTI service instance, time for the RTI service to invoke the RTIExec, and time to retrieve the multicast endpoint from the RTI service; (C) time to get the federate factory service handle from the index service; and (D) time to create a federate process, including time to connect to the federate factory service to create a federate service instance and time to connect to the federate service to create a federate process. The results are summarized in Table 3.2.

It should be noted that test case A takes significantly longer than the other cases because it incorporates the time for Globus to set up the local running environment. Test cases B1, B2 and B3 are in parallel because only one case will be encountered in the actual run of the simulation. Case B2 takes 2.3 seconds longer than B1 because it takes about that long to properly invoke the RTIExec. Case B3 would be the worst case for discovering the RTIExec multicast endpoint. The total time of using the Grid services to

Table 3.2: Test Results

Test case	Time (Seconds)
A	8.742
B1	0.350
B2	2.651
B3	4.501
C	0.151
D	0.989

setup a simulation is $A + B_x + C + D$, where B_x can be either B1, B2 or B3, depending on the setup of the Grid services. The overhead of the worst case ($A + B3 + C + D$) takes less than 15 seconds.

3.4 Summary

A framework for executing large-scale HLA-based distributed simulations with Grid services is developed in the preliminary study. The RTIExec control process is managed by the RTI service, which can be dynamically discovered. Simulation models are encapsulated within federate services so that the detailed implementations are hidden from end-users. These models are assembled through their Grid interface to form a large-scale distributed simulation. As different models can be dynamically located, it provides great flexibility. Reusability is inherently provided by the nature of Grid services. As the communication between federates is still through the RTI, the efficiency in communication at run-time is not compromised.

In this experimental framework, we have successfully investigated the dynamic discovery and remote invocation of the simulation components. However, there are some defects in the framework that restrict the effectiveness of conducting simulations, which are caused by the nature of this design, which is a hybrid of the HLA and the Grid.

First of all, the federate processes are still communicating through the RTI at runtime and the Grid is only used for initialization of the simulation sessions. There are several shortcomings if the simulation components communicate through the RTI. One concerns the firewall issue. For example, if two components are behind different firewalls, they may not be able to communicate with each other. Also many implementations of the RTI provide only centralized algorithms for some management services such as time management. These centralized algorithms will reduce the performance of the system especially when we are going to conduct a large-scale distributed simulation session.

The second problem is the inflexibility of the framework at runtime. As the federate process is only created by its federate service through the Grid, it is not possible for one federate process to request the creation of another at runtime in the current framework. The reason is that the federate process can only access the RTI layer but not the Grid layer at runtime. This limitation means that after the simulation session starts, the number of simulation components is fixed unless the client interferes in the simulation session.

Another problem is that existing federate processes are lacking a universal description language in the Grid layer. The HLA provides an interface definition file, i.e. the Simulation Object Model (SOM), to describe a federate but the service discovery is conducted over the Grid. The index service only contains very limited information of each simulation component, e.g. the address of the simulation component, which makes it very difficult for a client to choose suitable simulation components to form a simulation session.

To solve the problems listed above, a new simulation framework that is purely designed for the Grid is proposed. This is designed as a component-based, distributed simulation framework. The details of the design are presented in the next chapter.

Chapter 4

A Framework for Component-based Distributed Simulation on the Grid (SOAr-DSGrid)

4.1 Principles of the Design

With the experience gained from the preliminary study, we present the design of a new framework that is able to overcome the shortcomings that exist in the previous study. The aim is to develop a component-based, distributed simulation framework that provides users with easy design and deployment of simulation systems in a Grid environment. The following are some principles used in the design of the framework.

The complexity of the framework must be as simple as possible. A more complex framework means that it needs more effort to build the framework and more cost to maintain the framework. The current version of our framework is designed as a Service-Oriented Architecture. There are only four kinds of Grid services, i.e. simulation component, organizer, index service and repository (see 4.3 for more details).

The cost of developing simulations on the framework must be minimized. The cost of developing a simulation includes the design phase, the implementation phase and the results analysis phase. As our framework is component-based, each simulation

component can be reused in other simulation sessions and several simulation components are able to form a composite component (see 4.2.1 for more details). This is expected to reduce the cost of design and implementation of simulations by reusing existing components.

The efficiency of the framework must be maximized. In the framework design, we are mainly concerned with the efficiency of the data flow and control flow. Data flow describes the way that data moves in the system while control flow determines the order of activities that happen in the system. In order to design a distributed framework, both flows are designed to be distributed so that there is less chance to cause a traffic bottleneck in the framework. We develop a local time manager and a local rule engine to enable both flows to be distributed (see 4.4 for more details).

In the development of the framework, the above principles are always followed so that the framework is in accordance with the objectives defined in the first chapter. The following sections present details of the design of the framework.

4.2 Concepts of the Framework

In many commercial simulation packages such as Arena [3], a simulation view that consists of entities, variables, queues and resources is presented. Entities are used to represent real things in the system. They move around, change status, affect and are affected by other entities. Attributes are the characteristics of all entities. The same type of entities have the same attributes but different values, e.g. time of arrival, priority etc. Variables reflect a characteristic of the system, regardless of entities, but they may be changed by entities. Resources refer to the services that can process entities, like equipment or machines. Usually an entity will seize a resource, use it, and then release it. Queues are the places for entities to wait when they cannot move on, for example, when the resource

they want to seize is unavailable. This view is also adopted in our framework.

In our framework, components, entities, rules and variables are the basic concepts. The resources and queues are implemented inside components. Rules enable distributed data and control flow. Details of these concepts are described in the following subsections.

4.2.1 Components

Components are services that have the capabilities to create or process entities (see 4.2.2). Each component possesses a set of variables (see 4.2.4), which indicate the current state of the component. A component can have one or more operations. Operations refer to the functionalities that the components can use to process the entities. For instance, a possible operation for a component which simulates an oil rig is drilling oil. The component can switch among different operations when it is required. The operation rule (see 4.2.3) of the component decides which operation to use to process the entities. The operation rule is usually specified by the client before the simulation session starts.

The component can be either a primitive or composite component. A primitive component is an atomic component, which is designed to accomplish operations independently. On the other hand, a composite component, or parent component, is composed of a group of other components, children components. The children components can be either primitive or composite components. Each composite component has a composite component container, which sets up the composite component. The composite component container is also responsible for the coordination between the children components at runtime.

4.2.2 Entities

The entities are the objects that are created or processed by components. Each entity has a set of attributes that indicates the state of the entity. In a simulation session, entities

are transported between different components. The routes that the entities follow are defined by the routing rule (see 4.2.3) in each component. In a simulation, entities can represent raw materials, like stone or meat; they also can be in the form of final goods like cars or airplanes.

4.2.3 Rules

There are two types of rules in the framework, namely operation rules and routing rules. Operation rules are the decision logic to determine which operation will be used to process which entity when there is more than one available operation and entity for the component. The routing rules will be applied to determine which route will be chosen when there is more than one possible route for the processed entity to take. Normally a component has both an operation rule and a routing rule in a simulation session. Both of the rules for a component can be changed to meet the different needs of different simulation sessions.

4.2.4 Variables

Variables hold the information that describes the current state of the component that is the owner of the variables. There are two kinds of variables, namely public and private. Public variables are published and shared to other components. The owner of the public variables is responsible for the publishing and updating of the variables. Public variables are the main channels for components to communicate with each other besides the transferring of entities. Private variables are only accessible from inside the component.

Figure 4.1 demonstrates a possible simulation scenario that is composed of three components.

Figure 4.1: Concepts of the Framework

In Figure 4.1, component A connects to both component B and component C. Routing rule A1 decides to which component the entities shall flow. Operation Rule A1 decides which operation shall be used to process the entities that flow into component A. The public variables set A is owned and published by component A. It is also subscribed by component B so that whenever there is a change to the variables, component B will be informed.

In this thesis, we use the term shared variable to mean a public variable of another component that is subscribed by the current component. Thus the public variables set A of component A are shared variables of component B.

4.3 Overview of the Framework

From the concepts described in the last section, a framework is designed and developed. In Figure 4.2, there are six main parts, service provider, client, simulation components, organizer, index service and repository where the last four parts are implemented as Grid services. Their primary roles are introduced in the following sections.

Figure 4.2: Simulation Framework

4.3.1 Service Provider

The main role of service providers is to design, implement and deploy the simulation components for the framework. In addition, they also need to maintain and upgrade the simulation components if necessary. The service provider has to submit a component interface description file to the repository for the component that it developed. The file defines the interface of a component including information about public variables and operations etc. Clients are able to find suitable components for their simulation session by examining component interface description files. The file format is the same for both primitive components and composite components. The difference is given by the value of the “isComposite” tag in the file. The full format of the file is provided in Appendix A.

As mentioned above, the component interface of a composite component is defined by the component interface description file. However how the internal components of a composite component interact with each other is unknown from this file. This information is recorded in the composite component implementation file. The file contains the information of the primitive components and composite components that constitute the composite component. In addition, it also illustrates their relationships at runtime. The full format is available in Appendix B.

4.3.2 Client

Clients are those who want to make use of this framework to run simulation sessions. They are responsible for the design of the simulation session. For instance, the issues of the design are “what is the purpose of the simulation”, “what are the input parameters”, “what components are used in the simulation” etc. After the client has designed the simulation session, he has to compile the simulation description file, which contains the detailed information of the simulation session. The structure of the file is listed in

Appendix C. The simulation session will start after the client submits the simulation description file to the organizer (see Section 4.3.4).

4.3.3 Simulation Component

A simulation component is designed to simulate particular objects by a service provider e.g. a machine in a factory's simulation. A simulation component can be either a primitive component or a composite component. A primitive component is an independent process which can simulate an object. A composite component, on the other hand, is composed of several primitive components and/or composite components to form a more complex component. An example of a composite component structure is shown in Figure 4.3; it includes a composite component container and several primitive components. The real work will be performed by the primitive components and the composite component container is mainly responsible for coordinating the running of the primitive components and maintaining the composite component attributes. Due to limitations of time, composite components are not yet implemented in this project.

Figure 4.3: Composite Component Structure

4.3.4 Organizer

An organizer is used to initiate and terminate the whole simulation session. Before starting the simulation session, the client will send the simulation description file to the organizer. Then the organizer will parse the file to get the information about all the simulation components that will run in this simulation session. After initialization finishes, the organizer will run in the background and monitor the running of the simulation session. During the running of the simulation session, the client is able to call the organizer to stop the simulation if the client feels that this is necessary. At the end of the simulation, the organizer will send the results of the simulation back to the client.

4.3.5 Index Service

The index service is used to store information about the framework, i.e. the address of different types of components and the repository. Clients query the index service to find components to design the simulation. The organizer queries the index service to locate simulation components and the repository for component interface description files and other interface files to organize a simulation at runtime.

4.3.6 Repository

The repository is used to store five kinds of file, namely component interface description files, composite component implementation files, entity type files, rule type files and rule files. Entity type files specify the data structure of entities. An example of an entity type file is given in Appendix D. Rule type files are used by the component manager to create input data for rules (see Section 5.3.1). Rule files are used to store the implementation of rules. The service providers upload all these files into the repository when they deploy the corresponding simulation component. In addition, clients are also able to upload rule

type files and rule files that are developed by them for a particular simulation component to the repository .

4.4 Design of Primitive Component

Figure 4.4: Primitive Component Structure

Figure 4.4 shows the structure of a primitive component in our framework. From Figure 4.4, we can see that a component communicates with other components by means of either entities or shared variables. Entities received from other components are stored in the external entity queue and internal entity queue. When the simulation is running, the available operations are called by the component manager to process the entities; the rule engine makes decisions about which entity is to be processed next, which operation is to be used to process the entity and where to send out the processed entity. The processed entities are sent out by the output entity manager. The time manager manages the logical time. In the rest of this section, the supporting parts are introduced.

4.4.1 Component Manager

The component manager will be the first part of a simulation component to be executed when the organizer initializes the simulation component. In the set-up stage, the component manager will initialize other parts of the component, e.g. shared variable manager, rule engine etc. After initialization completes, there will be two active threads, i.e. component manager and shared variable manager. A primitive component can be either an entity generator or an entity processor. An entity generator actively generates entities and send generated entities to other components while an entity processor starts to process entities after it receives them from other components. The component structure is identical for them, but the procedure for the component manager to execute them is different. Figure 4.5 shows the interactions between the component manager, operation, rule engines and output queue at runtime when the component is an entity generator. Figure 4.6 show the interactions between the component manager, internal entity queue, operation, rule engines and output queue at runtime when the component is an entity processor.

4.4.2 Operations

Operations are developed by service providers. They are called by the component manager to process entities received from others or to generate new entities. Figure 4.7 illustrates the design of the abstract operation class. There are some situations that require an operation to process more than one entity at the same time, e.g. some tools of wafer fabrication can process a set of batches at the same time. A component can have more than one operation which enables a component to process different entity types, but only one operation can be active at any time. The component manager is in charge of switching the operation according to the type of entities.

Figure 4.5: Flowchart of Component Manager of an Entity Generator

Figure 4.6: Flowchart of Component Manager of an Entity Processor

```
//Abstract Operation Class
public abstract class Operation {

    //initialize the operation, invoked when the object
    //is instantiated
    public void initializeOperation(){}

    //call this method before call execution
    //used to acquire resource
    public void preExecution(Hashtable inputEntities){}

    //execution method
    public void execute(Hashtable inputEntities) {}

    //postExecution
    //called by time manager after time advanced
    //used to release resource
    //no input parameters
    public void postExecution(){}

}
```

Figure 4.7: Abstract Operation Class

There are four methods.

- initializeOperation()

This method is called by the component manager to initialize an operation. This usually includes initializing resources owned by this operation. It is called only once for each simulation session.

- preExecution(Hashtable inputEntitis)

This method is called to acquire necessary resources to execute the operation. It is called before execution. The input parameter is the set of entities to be processed by the operation.

- execute(Hashtable inputEntities)

This method is called by the component manager to execute the operation. The input parameter is the set of entities to be processed by the operation.

- postExecution()

This method is called to release the resources acquired by `preExecution()` when the time manager grants the time advancement requested by the internal event that represents the end of processing (see Section 4.4.4).

4.4.3 Entity Queues

There are three entity queues in the framework, i.e. external entity queue, internal entity queue and output entity queue. The external entity queue is used to store entities just received from other components. The entities on this queue cannot be processed immediately. The local time manager will move an entity from the external entity queue to the internal entity queue when it meets the conditions for delivery (see Section 5.1). The internal entity queue stores entities that are ready to be processed by the component. They will be removed from the internal entity queue when they are processed. All the processed entities or new entities will be put on the output entity queue where they will wait to be sent out. All three entity queues are sorted by timestamp order, however the processing sequence may not follow this order.

4.4.4 Internal Event Queue

The internal event queue is used to store events generated by operations. After processing entities, the operation will generate an event to indicate how much simulated time is spent on processing those entities. This event is then put on the internal event queue. The timestamp stored in the event is one of three time values that are used to decide how much logical time the simulation can advance. The other two time values are the nearest time in the future that an entity can be received from other components and the timestamp of the first entity on the external entity queue (see Section 5.1).

4.4.5 Local Rule Engine and Rules

There are two types of rules in the framework, i.e. operation rules and routing rules.

- Operation Rules

When the component manager wants to process entities on the internal entity queue, it will consult the local rule engine. The operation rule decides the processing order of entities. The local rule engine will read in all the entities on the internal entity queue and then choose entities that can currently be processed, together with the appropriate operations for those entities.

- Routing Rules

After entities have been processed by operations, the component manager has to choose the correct target simulation components to which to send them. The component manager consults the local rule engine. The routing rule is used by the local rule engine to make the decision.

One of the main objectives of the framework is to decentralize the data flow and control flow. The benefit of a distributed architecture is that we can minimize the bottleneck which occurs in a large centralized architecture. Entities and the values of shared variables inside the framework are transferred in a point-to-point manner, so the data flow is distributed. Both operation rules and routing rules are designed to enable distributed data flow and control flow in the framework. The operation rule selects the appropriate operation to process input entities and the routing rule chooses the destination components for output entities.

The local rule engine is responsible for the proper initialization of these two types of rules. It exists in every simulation component in the framework. In the initialization of the simulation session, the organizer will transfer the name of the operation rule and

routing rule to the component. The component then downloads the rule type files and rule files from the repository.

4.4.6 Shared Variable Manager

The shared variable manager manages shared variables in the simulation component. A shared variable is *updated* by its owner and can be *queried* by other simulation components. When a shared variable is queried by another simulation component for the first time, the shared variable manager will record the simulation component as a subscriber. Once the value is updated by its owner, the shared variable manager has two options, i.e. sending the value to its subscribers (*PUSH*) or waiting to be queried (*PULL*). *PUSH* is preferred when the value is required frequently by the subscribing component because this can effectively reduce repeating queries from the same simulation component. On the other hand, *PULL* is preferred when the value is queried infrequently [22]. A history list and a future list are used in the shared variable manager. Because of the use of the history and future lists, time advancement and shared variable support can be handled separately without local causality violations. Once the values in the history list and future list become obsolete, they will be purged by the shared variable manager. Table 4.1 shows an example of the future list stored in component A assuming that the current logical time is 5. Table 4.2 shows an example of the history list of component A.

Table 4.1: Future List stored in Component A

VariableOwner	VariableName	Value	TimeStamp
ComponentD	QueueLength	20	9.00
ComponentC	QueueLength	25	10.00
ComponentC	QueueLength	28	12.00
ComponentD	QueueLength	29	12.00
ComponentE	QueueLength	15	12.00

Table 4.2: History List stored in Component A

VariableOwner	VariableName	Value	TimeStamp
ComponentA	QueueLength	14	1.00
ComponentA	QueueLength	22	2.00
ComponentA	QueueLength	6	3.00
ComponentA	QueueLength	9	4.00

In Table 4.1, we can see that the queue length of component D at logical time 9 is 20 and it changes to 29 at logical time 12. This change happens in the future of component A as the current time of component A is only 5. Table 4.2 shows records of the queue length of component A as it changes from logical time 1 to 4.

4.4.7 Local Time Manager

As the current SOAr-DSGrid is designed to conduct event-driven simulations, the time manager currently supports only the *Next Event Request* (NER) [21]. In our framework, the *Next Event Request* has no parameter. It tries to advance the current time to a certain logical time T that allows entities on the internal entity queue to be safely processed. The details of the local time manager are given in the next chapter.

4.5 Executing a Simulation Session

Figure 4.8 illustrates how to start a simulation session as a client. There are three stages for the client program in a simulation session, i.e. initialization, runtime and termination. The client program can be implemented in any language that can communicate with the index service and organizer. However, the Java language is recommended as the whole framework is developed using the Java language. The Java pseudo code for these procedures is listed in Figure 4.9. From Figure 4.9, we can see the detailed procedure to

Figure 4.8: Involvement of Client Program

execute a simulation is as follows:

- Query a well-known index service for an organizer.
- Submit the simulation description file to the organizer.
- Start the simulation.
- Monitor the status of the simulation and wait for it to finish.
- Collect the simulation results.


```

Initialization {
//query for the address of organizer factory
organizerHandle=indexService.query("organizer");
//instantiate an organizer
organizer=createService(organizerHandle);
//submit the simulation description file
organizer.submitFile(simulationDescriptionFile);
Runtime();
}

Runtime() {
    for(;;)
    {
        //waiting for the simulation finish
        sleep(N);
        //query for the status of simulation
        status=organizer.getSimulationStatus();
        if (status == abnormal){
            //terminate the running if simulation goes wrong
            organizer.terminate();
            quit;
        }
        if (status == over){
            Termination();
            quit;
        }
    }
}

Termination() {
//retrieve result
result= organizer.getResult();
//terminate the simulation
organizer.terminate();
}

```

Figure 4.9: Pseudo Code of Client

4.6 Summary

In this chapter, the design details of the framework for component-based distributed simulation on the Grid (SOAr-DSGrid) are presented. First, the design principles of the framework are introduced. They can be summarized as making the complexity of the framework as simple as possible, minimizing the cost of developing simulations on the framework and maximizing the efficiency of the framework.

The four basic concepts of the framework are components, entities, rules and variables.

Components are the services that create and process entities. Rules are used to instruct components how to process entities and where to send processed entities. Variables store the state of components.

The framework requires six parts to conduct a simulation, i.e. service provider, client, simulation components, organizer, index service and repository. Service providers are responsible for developing simulation components. To start a simulation session, the client needs to query the index service to find the organizer. The organizer helps the client to set up the simulation using the simulation components. After the simulation ends, the organizer will return the results to the client. The repository is used to store component interface description files, composite component implementation files, entity type files, rule type files and rule files.

There are two types of simulation component, primitive and composite. Details of how the primitive component is designed and implemented are described. A primitive component consists of the component manager, operations, entity queues, internal event queue, rule engine, shared variable manager and local time manager. A composite component is composed of primitive components and composite components. Due to time limitations, composite components are not yet implemented in this project.

In the next chapter, execution support for the framework is introduced. The execution support includes the local time manager and rule engine.

Chapter 5

Execution Support

5.1 Fully Distributed Simulation Execution

Fully distributed simulation execution in our framework means that there should be no centralized communication that may constrain the scalability of the system when the simulation is running. In other words, if a component needs to communicate with another component, the communication must happen in a peer-to-peer manner without the participation of any centralized service. There are three major activities that involve communication between components, i.e. time advancement, shared variables and transferring entities between components. A local time manager is developed to achieve distributed time management. A shared variable manager is implemented to achieve distributed shared variable management^{5.1}. A local rule engine is developed to enable distributed data flow and control flow. The local time manager and local rule engine are introduced in the following sections.

^{5.1}The shared variable manager was developed with another masters candidate Chen Xinjun from PDCC, Nanyang Technological University. The implementation of the shared variable manager is not the focus of this thesis.

5.2 Local Time Manager

The time manager is designed to manage the logical time for the framework. The major objective of the time manager is to enable the distributed computation of the logical time of each component. The time manager is organized as a set of local time managers, one per component, and is thus referred to as a distributed time manager. In Section 5.2.2, we present the design of the local time manager but first we consider the interactions between components.

5.2.1 Interactions between Components

In the framework, simulation components interact with each other during simulation execution through sending entities and querying shared variables. Entities are defined as objects that can be created, processed and consumed by simulation components, while shared variables are used to store the states of simulation components and are public to others. This is different from a traditional parallel or distributed message-passing simulation system, where simulation components interact only by exchanging time-stamped messages.

In Figure 5.1, component C owns a set of shared variables, EntityProcessed and EntityGenerated. They represent the number of entities processed and generated by component C. Both component A and component D are interested in these two variables and hence subscribe to them. After subscribing to the variables, they will be able to get the values of the shared variables from component C.

Also in Figure 5.1, component C receives entities from components A and B, hence components A and B are component C's regulating components. Component C sends entities to components D and E, hence components D and E are component C's constrained components. Component C maintains tables of its regulating and constrained compo-

Figure 5.1: Interactions between Components

nents. We define that a component can send entities to components in its constrained components table and receive entities from components in its regulating components table. This topology information is used by the distributed time manager.

5.2.2 Design of the Local Time Manager

In the framework, the time manager works closely with several other managers and component resources, i.e. the internal event queue, the external and internal entity queues, operations, the regulating components table, and the constrained components table. Figure 5.2 illustrates the important dependency relationships between these different parts and the distributed time manager.

LBTS and Regulating Components Table

We define that a component's regulating components are those components that can directly send entities to it. A component stores the status of its regulating components

Figure 5.2: Relationship between Different Parts of a Component and the Distributed Time Manager

in the regulating components table. The Lower Bound Time Stamp (LBTS) is defined as the nearest time in the future that an entity can be received from its regulating components. As only components in the regulating components table are able to send entities to the component, the time manager can calculate T_{LBTS} by simply looking at this local table. The value of each entry in the regulating components table is updated by the corresponding component through reliable and FIFO Grid invocation so that the values in the regulating components table are always synchronized with the original values. The Grid interface for this invocation is maintained by the component manager. Table 5.1 is a snapshot of component C's regulating component table. The value of the component time in the table is a lower bound on the logical time that Component C can receive entities from Components A and B respectively. The relationship of components is shown in Figure 5.1. T_{LBTS} is the minimum value of the times in the regulating component table which is 20 in the example given in Table 5.1.

Table 5.1: Regulating Components Table stored in Component C

Component Name	Component Time
Component A	20
Component B	25

Lookahead and Constrained Components Table

On the other hand, the constrained components are defined as those components to which the current component can directly send entities. Lookahead is a guarantee from a simulation component that it will not generate any external message with a timestamp smaller than its current time plus the value of lookahead [21]. In our framework, lookahead is defined as the transfer time of an entity from one simulation component to another. Every simulation component maintains a constrained components table that stores the lookahead values to its constrained components. If simulation component A's current time is T_c and its lookahead for component B is L , then any entity sent from component A will not reach component B before $T_c + L$. Table 5.2 give an example of the constrained components table stored in component C. It indicates that the lookahead values from component C to component D and component E are 10 and 12 respectively.

Table 5.2: Constrained Components Table stored in Component C

Component Name	Lookahead Value
Component D	10
Component E	12

In the framework, we assume that there is no cycle of simulation components where the lookahead value between every pair of neighboring components in the cycle is zero. This is necessary because once a simulation component advances its logical time to T' ,

it is guaranteed by the distributed time manager that the component will not receive in the future any entity with a timestamp less than or equal to T' . If there exists a cycle of zero lookahead, the guarantee will be violated as the component may receive an entity from another component at T' . This will result in the possibility of deadlock and loss of repeatability.

External Entity Queue and Internal Entity Queue

The external entity queue stores the entities just received from other simulation components. The smallest timestamp of entities in the external entity queue should be greater than the simulation component's current logical time T . When the simulation component is granted a time advancement to logical time T' , entities in the external entities queue with timestamp less than or equal to T' will be moved to the internal entity queue. Entities in the internal entity queue are safe to be processed by the simulation component.

Internal Events and Operations

Internal events are generated by operations to indicate the finish time of the operation. An operation can only be processed if its required resources are available. The operation will generate an event with a timestamp equal to the current time plus the simulation time needed to process the entity. The event is stored in the internal event queue. When the event is processed, the resource used by the operation will be released.

5.2.3 Implementation of the Local Time Manager

The distributed time management algorithm involves communication of entities and other information on the network. Communication between components is assumed to be reliable. This means that there will be no lost entities or messages. The generic component data structures used in the framework are shown in Figure 5.3, Figure 5.4 and Figure 5.5.


```

//Entity Class
class Entity{
    //type
    String type;
    //processing time for this entity
    //estimated by the operation
    double processTime;
    //creator of this entity
    String creator;
    //the time that the entity is created
    double createdTime=0;
    //the time that the entity is received
    double rcvTime=0;
    //return rcvTime
    double getTime();
    //constructor for the class
    public Entity(createdTime, creator, type);
}

//Entity Queue Class
//Queue stores Entities
class EntityQueue{
    //current size of entity queue
    int size;
    //insert new entities
    void insert(Entity entity);
    //get entity by sequence number
    Entity get(int ID);
    //get first entity on the queue
    Entity getFirst();
    //remove entity
    void remove(Entity entity);
    //remove entity by sequence number
    void remove(int ID);
    //check if the queue is empty
    boolean isEmpty();
}

```

Figure 5.3: Entity and Entity Queue Classes

The Entity class (Figure 5.3) is the generic data structure used to store the state of an entity. It includes the type, processTime, creator, createdTime and rcvTime. The processTime is estimated and set by the component manager before the entity is scheduled into the operation. The createdTime refers to the time when the entity is placed on the outputEntityQueue. The rcvTime indicates the time that the entity will be received by the next component. It is computed by adding the createdTime and the lookahead value between this component and the next component. The EntityQueue class is used for queues that store entities. There are three EntityQueues in a component,

```

//Event Class
class Event{
    //type
    String type;
    //the time that the event will happen
    double happenTime=0;
    //return happen time
    double getTime();
    //constructor for the class
    public Event(happenTime, type);
}

//Event Queue Class
//Queue stores Events
class EventQueue{
    //current size of event queue
    int size;
    //insert new event
    void insert(Event event);
    //get event by sequence number
    Event get(int ID);
    //get first event on the queue
    Event getFirst();
    //remove event
    void remove(Event event);
    //remove event by sequence number
    void remove(int ID);
    //check if the queue is empty
    boolean isEmpty();
}

```

Figure 5.4: Event and Event Queue Classes

i.e. `externalEntityQueue`, `internalEntityQueue`, `outputEntityQueue` (see Section 4.4.3). The `EntityQueue` class provides several generic methods, e.g. `insert()`, `get()`, `getFirst()`, `remove()` and `isEmpty()`.

The `Event` class and the `EventQueue` class (Figure 5.4) are similar to the `Entity` and the `EntityQueue` class, however `EventQueue` is only used to store internal events. The `Component` class (Figure 5.5) is the generic data structure used to store the states of a simulation component. It includes the name of a component and its component time. There are two other classes that extend it, i.e. `RegulatingComponent` class and `ConstrainedComponent` class. A `RegulatingComponent` class stores the component time of a regulating component which is the current time of the regulating component plus the lookahead between that component and the current component. A `ConstrainedCompo-`

```

//Component Class
//Used to store states of a component
//If the component object refers to the states of another component,
//then those states are updated by that component through the shared
//variable manager.
class Component{
    //name of the component
    String name;
    //current time
    double currentTime;
    //set current time
    double setTime();
    //return current time
    double getTime();
}

class RegulatingComponent extends Component {
    //set component time, overrides setTime() in Component class
    double setTime();
    //return component time, overrides getTime() in Component class
    double getTime();
}

class ConstrainedComponent extends Component {
    //lookahead to this component
    double lookahead;
    //return look ahead
    double getLookahead();
}

```

Figure 5.5: Component Class

nent class stores the lookahead value between the current component and a constrained component.

Calculation of LBTS

As mentioned previously, LBTS is the Lower Bound Time Stamp on entities that can be received from other components. The capability to calculate LBTS in a distributed manner is the key in building the distributed time manager. Thus, each simulation component maintains in its regulating components table the logical time of each regulating component plus the lookahead value between the regulating component and the component itself. Figure 5.6 demonstrates how to use this table to obtain a component's LBTS.

```
//Time Manager class
public double caculateLBTS(){
    for(i=0;i<regulatingComponents.size;i++){
        component=regulatingComponents[i];
        if (LBTS==NULL||LBTS>component.getTime())
            LBTS=component.getTime();
    }
    return LBTS;
}
```

Figure 5.6: Calculation of LBTS

It is clear that the correctness of LBTS relies on the correctness of the values in the regulating components table. To make sure the regulating components table is always valid, regulating components will inform the component whenever they advance their time. Figure 5.7 shows this action.

```
//Time Manager class
public synchronized void informTimeAdvanced(){
    for(i=0;i<constrainedComponents.size;i++){
        component=constrainedComponents[i];
        time=currentTime+component.getLookahead();
        //getLocalComponent() returns
        //an identifier to the component itself
        updateTimeAdvancement(time, getLocalComponent());
    }
}
```

Figure 5.7: Inform Time Advanced

The corresponding method that processes this information is the updateTimeAdvancement method as shown in Figure 5.8. This method will also try to wake up the component manager thread in the case where the time manager is in pending status waiting for an NER to be granted. After being woken up, the distributed time manager will execute another iteration of the NER algorithm (see next section).

```

//Time Manager class
public synchronized void updateTimeAdvancement(double time,
    Component regulatingComponent){
    if(regulatingComponent.getTime()<time){
        setTime(regulatingComponent, time);
    }
    if(status==pending){
        notifyAll(); //wake up the thread
    }
}

```

Figure 5.8: Update Time Advancement of Constrained Component

Next Event Request

In one iteration of the Next Event Request, the time manager must decide if the request can be granted, an entity can be processed or the NER has to wait. To make the decision, the time manager must check (i) the timestamp of the first event on the internal event queue (T_{Event}), (ii) the receive time of the first entity on the external entity queue (T_{Entity}), and (iii) the Lower Bound Time Stamp (T_{LBTS}). If there is no event on the internal event queue, or no entity on the external entity queue, then T_{Event} or T_{Entity} will be set to positive infinity. There are six possible combinations of these variables as shown in Figure 5.9. They can be further grouped to 3 cases.

- Case 1, NER grants event processing. In this case, T_{Event} is smaller than T_{LBTS} and smaller than or equal to T_{Entity} . It means that the simulation component has to process the event(s) with the T_{Event} timestamp first. NER will call the appropriate operation(s) to process the event(s) on the internal event queue with timestamp T_{Event} . Logical time advances to T_{Event} .
- Case 2, NER grants entity processing. T_{Entity} is smaller than T_{LBTS} and smaller than or equal to T_{Event} . The NER will deliver entities with timestamp equal to T_{Entity} on the external entity queue to the internal entity queue. Logical time advances to T_{Entity} . NER will allow the component manager to process entities on

Figure 5.9: Three NER Cases

the internal entity queue.

- Case 3, NER waits. T_{LBTS} is the smallest among the three time variables. This implies that the regulating components can still possibly send entities with timestamp T_{LBTS} . It will be unsafe to grant the NER. In this case, the NER has to wait. However, logical time still advances to T_{LBTS} .

If T_{Event} is equal to T_{Entity} and both are less than T_{LBTS} , case 1 and case 2 will both be executed but case 1 will be executed first.

The pseudo code shown in Figure 5.10 implements the time management algorithm outlined above. In the implementation, we introduce a variable status to indicate the status of the time manager, namely pending or idle. Pending means that the NER has not been granted and it is not safe for the component to process any entity whereas idle indicates that the component can process the next entity on the internal entity queue.

Once logical time is advanced, the NER will inform its constrained components about the time advance. This is necessary as its constrained components are possibly pending and waiting for this information. At the end of each iteration of the NER algorithm, if the status is still pending, the component manager thread that calls the NER method will block otherwise the NER is granted.

If the NER is pending, the blocked component manager thread will be woken up if one of its regulating components executes the `updateTimeAdvancement()` method. After the thread is woken up, the NER algorithm will start a new iteration. Figure 5.11 shows an example of interactions between components regarding the pending and waking actions at runtime.

```

//Time Manager Class
public synchronized void nextEventRequest(){
    do{
        status=pending;
        LBTS=caculateLBTS();
        minTime=minimum(LBTS,eventHead,entityHead);
        if(minTime==eventHead && eventHead<LBTS){
            logicalTime=eventHead;
            event=internalEventQueue.getFirst();
            do(){
                Process(event);
                internalEventQueue.remove(event);
                event=internalEventQueue.getFirst();
            }while(event.getTime()==minTime);
            status=idle;
        }
        if(minTime==entityHead && entityHead<LBTS){
            logicalTime=entityHead;
            entity=externalEntityQueue.getFirst();
            do(){
                internalEntityQueue.insert(entity);
                externalEntityQueue.remove(entity);
                entity=externalEntityQueue.getFirst();
            }while(entity.getTime()==minTime);
            status=idle;
        }
        if(minTime==LBTS){
            logicalTime=LBTS;
            status=pending;
        }
        informTimeAdvanced();
        if(status==pending){
            wait();
        }
    }while(status==pending);
}

```

Figure 5.10: NER Algorithm

5.3 Local Rule Engine and Rules

There are two types of rules existing in the framework: operation rules and routing rules. An operation rule is designed to select the proper entity to be processed and operation to be used based on available component resources. A component resource can refer to any variable state of a component. For example, in a car park simulation component, its component resources will be the car park slots. The operation rule in this situation will decide which car can get a car park slot. A routing rule on the other hand decides to which component the processed entity will be sent. For instance, suppose in a manufacturing

Figure 5.11: Pending and Waking

simulation, after a TV set is produced, there are two warehouses available. The routing rule will decide to which warehouse the TV set will be sent. A component cannot change its operation rule and routing rule after the simulation session starts. However, clients can change the rule easily for the next simulation session if necessary.

The local rule engine is used to execute the rules. There are two instances of the local rule engine, one is for the operation rule and the other is for the routing rule.

5.3.1 Design of Operation Rule and Routing Rule

Rule Type File

Figure 5.12 shows an example of a rule type file in XML for the FIFO operation rule and shortest queue routing rule. It tells the component manager that for the FIFO operation rule, the input parameters are resource and internalEntityQueue. resource is an integer

```

<Rules>
  <Rule Name="FIFO" Type="Operation">
    <Variable Name="resource" Type="integer">
      <Component Type="Self">
      </Component>
    </Variable>
    <Variable Name="internalEntityQueue" Type="EntityQueue">
      <Component Type="Self">
      </Component>
    </Variable>
  </Rule>
  <Rule Name="Shortest Queue" Type="Routing">
    <Variable Name="internalEntityQueueSize" Type="integer">
      <Component Type="Constrained">
      </Component>
    </Variable>
    <Variable Name="outputEntityQueue" Type="EntityQueue">
      <Component Type="Self">
      </Component>
    </Variable>
  </Rule>
</Rules>

```

Figure 5.12: Rule Type File

and `internalEntityQueue` is `EntityQueue` type. It also tells the rule engine that for the shortest queue routing rule, there are two input parameters. One is `internalEntityQueueSize`, which is integer type and one is `outputEntityQueue`, which is `EntityQueue` type. The component manager asks the shared variable manager to retrieve the `internalEntityQueueSize` from all its constrained components according to the XML file and then the component manager will store the values of `internalEntityQueueSize` in an array and pass them to the rule engine.

In addition, there are some default data structures which are always passed into the rule engine e.g. constrained components table and regulating components table, which contain the states of the constrained components and regulating components respectively.

BeanShell as Rules Interpreter

As an operation rule and a routing rule are invoked every simulation cycle, we need to find an interpreter that can execute our rule files efficiently. BeanShell is a small, embeddable Java source interpreter with object scripting language features, written in Java [38]. In addition, it dynamically executes standard Java syntax and extends it with

common scripting conveniences such as loose types, commands, and method closures like those in Perl and JavaScript.

Rules that are written in BeanShell look no different to any normal Java code. However, the following points should be noted.

- Library files for new classes used in the rule file must be included.
- All the input parameters can be directly used without definition in the rule files. However, it is good practice to list all input parameters as comments at the top of a rule file so that it is easier to maintain rule files.
- The return result must be named as “result”. It is a hashtable. For operation rules, the entry contains the entities and the corresponding operation to process them. For routing rules, each entry contains the entity and the corresponding component to which the entity is sent.

Figure 5.13 is a rule file that implements the FIFO operation rule for a component. The rule file will examine the entities on the internal entity queue from the head of the queue. We assume that the operation can process a number of entities of the same type at once. The number of entities that can be processed is decided by the number of resources stored in the variable resource. When resources are available, the rule will assign the first entity on the queue into the operation and set the current operation. Then the rule engine will keep looking for the same type of entity and schedule it into the operation until there are no free resources. The resources that are allocated to process the entities will be available again when the operation finishes processing the entities. Table 5.3 shows the hashtable that stores the result from the FIFO operation rule. It contains the operation name and a set of entities that will be processed by the operation.

Figure 5.14 shows a rule file that implements the shortest queue routing rule. This rule

```

include EntityQueue;
include Entity;
include Operation;
include OperationTable;

//***** Variable Definition *****/
//variables passed in from external
//EntityQueue internalEntityQueue;
//int resource;
//OperationTable operationTable
//result stores the entities and their processing operation
Hashtable result;
//entities stores a set of entities that are going to be processed
//by one operation
List entities=new List();
//***** Rule Starts Here *****/
//get the current operation
//it is determined by the first entity's type on the queue
String currentEntityType=internalEntityQueue.get(0).type;
Operation currentOperation=OperationTable.getOperation(currentEntityType);
for(int i=0;i<internalEntityQueue.size;i++){
    Entity entity=internalEntityQueue.get(i);
    String entityType=entity.type;
    if((operation.requestResource<resource)&&(currentEntityType==entityType)){
        entities.add(entity);
        resource-=currentOperation.requestResource;
    }
}
result.set(entities,currentOperation);

```

Figure 5.13: FIFO Rule File

will select the component in the constrained component table with the shortest current internal entity queue size. Once an entity is scheduled for that component, the queue size of that component will be increased by one to pretend that an entity has been received by the component already. This increment is made by the local rule engine to ensure that the rule will be executed correctly for the later entities. The change only affects a local copy of the shared variables, i.e. it will not change the real value of `internalEntityQueueSize` which is owned by its constrained components. For example, in Figure 5.1, assume that

Table 5.3: Result from the FIFO Operation Rule

Entity Set	Operation
entities[i]	operationName

Table 5.4: Result from the Shortest Queue Routing Rule

Entity Name	Component Name
Entity 3	Component D
Entity 4	Component E

by using the shortest queue routing rule, entity 3 will be sent to component D and entity 4 will be sent to component E. Table 5.4 shows the hashtable that stores the result after the rule is executed. Each entry contains an entity and a component name referring to the component to which that entity will be sent.

```
include EntityQueue;
include Entity;
include Component;

//***** Variable Definition *****/
//variables passed in from external
//Component [] ConstrainedComponents;
//this array stores the internal entity queue size
//of its constrained components, it is a shared variable
//int [] ConstrainedComponentsQueueSize;
//***** Rule Starts Here *****/
Component result;
int shortestQueue=Integer.Max_Value;
//local copy of ConstrainedComponentsSize
int[] localConstrainedComponentsQueueSize;
for(int i=0;i<ConstrainedComponentsQueueSize.size;i++){
    localConstrainedComponentsQueueSize[i]=
        ConstrainedComponentsQueueSize[i];
}
for(int j=0;j<outputEntityQueue.size;j++){
    Entity entity=outputEntityQueue.get(j);
    for(int i=0;i<ConstrainedComponents.size;i++){
        int queueSize=localConstrainedComponentsQueueSize[i];
        if(queueSize<shortestQueue){
            shortestQueue=queueSize;
            result.set(entity, ConstrainedComponents[i]);
            localConstrainedComponentsQueueSize[i]++;
        }
    }
}
}
```

Figure 5.14: Shortest Queue Rule File

5.3.2 Design of the Local Rule Engine

The local rule engine initializes the rule by binding input variable names and values. It also collects the result after BeanShell executes the rule.

Figure 5.15: Local Rule Engine

Figure 5.15 shows the relationship between the local rule engine and the component manager, rule initialization file, shared variable manager and rules.

The actual execution sequence is:

1. The component manager selects input parameters for a rule as specified in the rule type file and puts them into a hashtable, named `inputData`.
2. If the input parameters include shared variables, the component manager will get the value from the shared variable manager.
3. The component manager then invokes the local rule engine to execute a rule by transferring the `inputData`.
4. The local rule engine invokes BeanShell and passes the input parameters into it.

5. After BeanShell executes the rule, the local rule engine will collect the result and pass it back to the component manager.

5.3.3 Implementation of the Local Rule Engine

Figure 5.16 is the code segment that passes input data into BeanShell. The process is called variable binding. It binds the variable name and data value stored in each element of the hashtable, inputData, and passes them into BeanShell so that rules executing in BeanShell can access these variables. A hashtable structure is used to store the input data as it clearly shows the relationship between the name and the value of the variables.

```
//Local Rule Engine
//this method is called by Component Manager
//to pass input data to BeanShell
public void bindVariables( Hashtable inputData){
    try{
        while(more data in inputData){
            interpreter.set(inputData.variableName,inputData.variableValue);
        }
    }catch(Exception e){
        System.err.println(e);
    }
}
```

Figure 5.16: Variable Binding

Figure 5.17 is the code segment that implements how the local rule engine invokes BeanShell. The method selectComponent is called to execute a routing rule file and the method selectOperation is called to execute an operation rule file.

5.4 Summary

In this chapter, the design and implementation of the time manager and rule engine are presented. The time manager manages the logical time of the framework in a distributed manner. It provides a next event request (NER) service to the simulation component.

```
//Operation Rule
public Hashtable selectOperation(File ruleFile){
    Hashtable result;
    try{
        //ruleFile is the name of operating rule
        //interpreter.source will run ruleFile
        interpreter.source(ruleFile);
        //get result after interpreter.source finished
        result=(Hashtable) interpreter.get("result");
    }catch(Exception e){
        System.err.println("Exception: "+e);
    }
    return result;
}

//Routing Rule
public Hashtable selectComponent(File ruleFile){
    Hashtable result;
    try{
        //ruleFile is the name of routing rule
        //interpreter.source will run ruleFile
        interpreter.source(ruleFile);
        //get result after interpreter.source finished running
        result=(Hashtable) interpreter.get("result");
    }catch(Exception e){
        System.err.println("Exception: "+e);
    }
    return result;
}
```

Figure 5.17: Execution of Operation and Routing Rules

The simulation component invokes NER to advance its logical time in order to process new entities. The rule engine can process two types of rules: operation rules and routing rules. The operation rule is executed by the rule engine when the component manager wants to process entities. It decides which entities are processed by which operations. On the other hand, the routing rule is executed when the component manager wants to send out processed entities. It decides to which components the processed entities are sent.

In the next chapter, experiments that evaluate the performance of the time manager are discussed, and in Chapter 7, a case study on the use of the rule engine is described.

Chapter 6

Evaluation of the Local Time Manager

6.1 Experiment Setup

The objective of the experiments is to measure the performance of the distributed time manager compared to a traditional centralized time manager when running a large-scale simulation system.

Neither the centralized time manager in the DMSO RTI nor any other existing time manager system is comparable to the distributed time manager due to different network environments. Thus, a *centralized time manager* is developed within the framework for experimental purposes.

6.1.1 Centralized Time Manager

The centralized time manager is implemented as an independent GT4 Grid service. The original NER algorithm is modified accordingly so it will now retrieve T_{LBTs} from the centralized time manager. In addition, a component will inform its logical time advancement to a centralized time manager instead of its *constrained components*. The centralized time manager will update T_{LBTs} for a particular simulation component when

its regulating components inform the time manager about their time advancement. The new T_{LBTS} will be sent to the simulation component when the component's NER status is pending. Figure 6.1 shows an example of how the centralized time manager works with other simulation components in the framework.

Figure 6.1: Example of Centralized Time Service

6.1.2 Simulation Models

Super-Ping Model

A Super-Ping simulation model [51] is implemented for the experiments. The Super-Ping model is a generalization of a ping-pong simulation and consists of a number of ping objects (simulation components) connected together in a ring. Entities can be transferred

in either direction on the network. There are two variables used by the Super-Ping simulation. X is the total number of components, N is the number of entities assigned to each component. Figure 6.2 shows an example of a Super-Ping simulation session.

Figure 6.2: Super-Ping Model

For this model, we increase the number of entities in the system and meanwhile we also adjust the number of components.

N-Way Connection Model

The N-Way connection simulation model is designed to test the efficiency of the time manager as the number of connections that need to be handled by a component is varied.

Figure 6.3 shows the composition of such a model.

Figure 6.3: N-Way Connection Model

Suppose X is the total number of components in the network, Y is the number of incoming/outgoing connections per component and N is the number of entities that each component will generate. In the experiments, we fix X and vary Y and N to test the efficiency. In this model, we interconnect components so that $Component_i$ can receive entities from $Component_{i+1}$ to $Component_{i+Y(mod X)}$. In Figure 6.3, X is 5 and Y is 3. We can see that $Component_1$ receives entities from $Component_2$, $Component_3$ and $Component_4$.

Table 6.1: Testbed Environment

Processes	CPU Info	Memory Info	OS
Simulation Components	Dual Xeon 2.6GHz	1GB	RedHat Linux 7.3
Centralized Time Server	Dual Xeon 2.6GHz	1GB	RedHat Linux 7.3

6.1.3 Experimental Results

The cluster that is used to simulate the Super-Ping model is composed of 1 head node and 9 compute nodes. The specification for these nodes is listed in Table 6.1. Simulation components and the time server are executed on different compute nodes.

The results of the experiments are the average number of NERs granted and the average number of entities processed for a simulation component. We set each simulation session to run for two minutes real time and repeat five times for each case. The results are presented in Figure 6.4 and Figure 6.5.

From Figure 6.4 and Figure 6.5, we can see that increasing the number of entities causes the average number of NERs granted and entities processed to increase. If there are fewer entities in the network, it is more likely that some simulation components will wait because their external entity queues are empty. However, increasing the number of simulation components has the opposite effect, causing the average number of NERs and entities processed to decrease. This is because increasing the number of components without increasing the number of entities will again result in some simulation components waiting because their external entity queues are empty.

Comparing the performance of the distributed time manager and the centralized manager in these two figures, with fewer simulation components and entities, there is little difference between the distributed and centralized time managers for both the average number of NERs granted and the average number of entities processed. The centralized

Figure 6.4: Average Number of NERs Granted

time manager generally has slightly better performance than the distributed time manager with less than four simulation components. However, we can see that when there are more entities and simulation components, the distributed time manager shows a better performance for both the average number of NERs granted and the average number of entities processed, which indicates that the distributed time manager has better scalability than the centralized version. In addition, from these two figures, the average number of NERs is almost twice the average number of entities processed. This is reasonable as NERs are granted for both events on the internal event queue and entities on the external entities queue. Each operation on an entity will generate an internal event to indicate when the entity has been processed.

The cluster that is used to simulate the N-way connection model is composed of 1

Figure 6.5: Average Number of Entities Processed

head node and 15 compute nodes. Figure 6.6 shows the composition of the cluster. The operating system for the cluster is Red Hat Enterprise Linux ES release 4.

For this experiment, we fix the number of components to 10 as this enables us to observe the performance of the distributed time manager when there are a significant number of entities that need to be received and processed simultaneously. The results are shown in Figure 6.7 and Figure 6.8. We vary the number of connections to each component (Y Axis) from 1 to 9 and also the number of entities generated by each component (N Axis) from 2 to 20. Each simulation session is executed for two minutes real time and repeated five times.

Figure 6.7 shows the average number of NERs granted within a simulation run and

Figure 6.6: PDCBL Cluster

Figure 6.8 shows the average number of entities processed within a simulation run. From these two figures, we can see that for a fixed number of components, when there are fewer connections between components, the local time manager outperforms the centralized time manager. When there are more connections between components, the performance of the distributed time manager becomes worse. The reason is that the distributed time manager will always try to inform its constrained components about its time advance. When there are more links, a component needs to inform more constrained components when it advances its time. However, for a centralized algorithm, a component only needs to inform the centralized time manager once for the time advance and leave the calculation of LBTS to the centralized manager. If the LBTS value for a constrained

Figure 6.7: Average Number of NERs Granted

component does not change, the centralized time manager will not send any message to it. In this situation, the centralized time manager will have superior performance to the distributed time manager. From this experiment, we can conclude that for a given scale of simulation system, with an increased number of links between components, centralized time management will have better performance. In the extreme case when every component is connected to every other component, the local time manager in a component works like a centralized time manager, it will receive time advancement from all other components and has to send an update time advance message to all other components. In this case, if there are X components in the system, there will be X centralized time managers. This will lead to a worse performance than a single centralized time manager.

Figure 6.8: Average Number of Entities Processed

6.2 Summary

In this chapter, two sets of experiments are presented. The Super-Ping model is designed to evaluate the performance of the distributed time manager as the scale of the simulation system is varied. The experimental results are then compared to the results collected with a centralized time manager. From the Super-Ping experimental results, we find that when there are more than four components on the network, the distributed time manager shows better performance for both the average number of NERs granted and the average number of entities processed. The results imply that the distributed time

manager has better performance than the centralized time manager with an increased number of simulation components.

The N-way connection model is designed to measure the performance of the time manager when the number of connections between components is varied. The results show that when there are fewer connections between components, the distributed time manager outperforms the centralized time manager. However when there are more connections, the performance of distributed time manager becomes worse. These results imply that for a given number of simulation components, the distributed time manager will have better performance for a sparse topology with few connection between components.

In the next chapter, a case study on the use of the rule engine will be given.

Chapter 7

Case Study: Use of the Local Rule Engine

7.1 Multi-Recipes Wet Benches in the Wafer Fabrication Process

In this chapter, we present a simulation of multi-recipes wet benches in the wafer fabrication process as a case study on the use of the rule engine.

The wet bench is an important tool in the semiconductor manufacturing industry. A wet bench is composed of a number of baths which are capable of processing wafer lots, including layering, patterning, doping, and heat treatment. A recipe typically states the processing time in each bath and the order of visits to multiple baths. A wet bench normally can handle a large number of recipes. However, switching between different recipes can require extra processing time. Therefore it is important to schedule the processing order of lots as different recipes require different baths and involve different processing times. It is important that dispatching is handled with care so that the tool is efficiently utilized. In turn, a minimum cycle time is achieved. The cycle time is defined as the simulated time between generating a lot and the lot leaving the wet bench after being processed.

Figure 7.1: System Overview

Figure 7.1 shows an overview of the simulation of multi-recipes wet benches. We can see from Figure 7.1 that the simulation is composed of several types of components, namely generator, wet bench allocation component and batch processing component. A batch consists of two lots. The global dispatching algorithm decides to which batch processing component the batches are sent and the local dispatching algorithm decides the sequence in which batches are processed by the batch processing component. We illustrate how to implement the system using SOAr-DSGrid in the following subsections.

7.1.1 Generator and Wet Bench Allocation Component

To simplify the simulation system, we group the generator and wet bench allocation component into a single component. This component is used to generate batches of lots and then send them to the appropriate batch processing component. Global dispatching algorithms are expressed as routing rules for the wet bench allocation component and

are used to select the batch processing component for batches. There are typically three global dispatching algorithms, i.e. earliest start time (EST), earliest finish time (EFT) and same recipe (SR).

```
//Bath
class Bath{
    String name;
    boolean occupied;
    double processTime;
    double finishTime;
    Batch batchInProgress;
}

//Recipe
class Recipe{
    String name;
    Bath baths[];
}

//Batch Entity
class Batch extends Entity{
    Recipe recipe;
    int currentStep;
    //time to process the whole batch
    double processTime;
}
```

Figure 7.2: Bath, Recipe and Batch Entity Data Structure

In Figure 7.2, the application specific data structures for the wet benches simulation are presented. The class Bath stores the state of a bath. The state includes its name, whether it is idle or occupied, its processing time, the time at which the current batch finishes processing and a pointer to the batch. The class Recipe is used to store a recipe. It includes the name of that recipe and the baths involved in the recipe. The class Batch stores the state of a batch. It is extended from class Entity so it has all member variables in class Entity and it also has three additional state variables, recipe, currentStep and processTime. Variable recipe is the recipe used to process the batch, the currentStep records its execution step and the processTime is the total simulation time needed to process the recipe.

```
//array stores current time of its constrained component
double[] ConstrainedComponentsCurrentTime;

//array stores estimated queue time of constrained components
double[] ConstrainedComponentsQueueTime;

//array stores current recipe of constrained components
Recipe[] ConstrainedComponentsCurrentRecipe;
```

Figure 7.3: Shared Variables used in Global Dispatching Rules

Figure 7.3 lists three arrays of shared variables that are used by the global dispatching rules. `ConstrainedComponentsCurrentTime` stores the values of the current time of the constrained components. `ConstrainedComponentsQueueTime` stores the values of the estimated time that will be spent in the internal entity queue of the constrained components before a batch can be processed. It is set as the waiting time of the latest batch that is scheduled into the constrained component's wet bench. `ConstrainedComponentsCurrentRecipe` stores the current recipes processed by the constrained components. It is the recipe of the latest batch that is scheduled into the constrained component's wet bench.

Figure 7.4 shows the implementation of the batch generating operation. It produces a batch with random recipe between 1 and 34 at a random interval between 0 and 360. The type of the batch is the recipe to be used for the batch. There is no regulating components for a batch generator. When the batch generator generates a batch, it will advance the logical time that is used to generate that batch.

After the batches are generated, the component manager of the wet bench allocation component will consult the routing rule, which implements the global dispatching algorithm, to decide to which batch processing component each batch is sent. There are three routing rules available.

```

//Operation Batch Generator
//generating batch at an random interval between 0 - 360.
public class BatchGenerator extends Operation{
    Random random;
    //initialize the operation, invoked when the object
    //is instantiated
    public void initializeOperation(){
        random=new Random();
    }

    //Hashtable inputEntities is null for batch generator
    //there is no resource acquiring
    //preExecution contains no action
    public void preExecution(Hashtable inputEntities){}

    //inputEntities is null for batch generator
    public void execute(Hashtable inputEntities){
        //randomly pick a recipe between 1 to 34 as the recipe of batch
        //type here used to store the recipe type
        int type=random.nextInt(33)+1;
        //the time to produce the batch is between 0 to 360
        double timeSpent=random.nextDouble()*360;
        //batch to be produced
        Batch generatedBatch;
        //current time
        double currentTime=localTimeManager.getCurrentTime();
        //generate a batch
        generatedBatch=new Batch(currentTime, "batchGenerator", type);
        //put the generated batch onto outputEntityQueue
        outputEntityQueue.insert(generatedBatch);
        //put the time advance request onto internal entity queue
        Event event=new Event((currentTime+timeSpent), "time advance request");
        internalEventQueue.insertEvent(event);
    }

    //there is no resource releasing
    //postExecution contains no action
    public void postExecution(){}
}

```

Figure 7.4: Batch Generator

- Earliest Start Time:

The earliest start time (EST) algorithm will select the batch processing component with the earliest time that processing can start. Figure 7.5 illustrates the algorithm's pseudo code. The constrainedComponents table stores all the components that can receive batches from the local component. To calculate the start time for a constrained component, i.e. a wet bench, we add its lookahead time and estimated queue time to the current time of that constrained component. Lookahead time

is the travel time from the local component to the constrained component. The estimated queue time is equal to the queue time of the last batch scheduled into the wet bench. The wet bench with the earliest start time will be selected by this rule. Both the queue time and the current time are shared variables updated by that constrained component. For simplicity, we assume that all wet benches can handle all recipes. This assumption applies to the routing rules described in this subsection. In the experiments described in Section 7.2, individual wet benches are configured to handle different recipes.

```
//called by Component Manager when there are new entities in the output queue.
//Component constrainedComponents[];
//EntityQueue outputEntityQueue;
//array stores current time of the constrained component
//double [] constrainedComponentsCurrentTime;
//array stores estimated queue time of the constrained components
//double [] constrainedComponentsQueueTime;
float startTime=Double.max_value;

for (int j=0;j<outputEntityQueue.size;j++){
    Batch batch=(Batch) outputEntityQueue.get(j);
    for (int i=0;i<constrainedComponents.size;i++){
        if (startTime>constrainedComponentsCurrentTime[i] +
            constrainedComponentsQueueTime[i] +
            constrainedComponents[i].getLookahead()){
            startTime=constrainedComponentsCurrentTime[i] +
                constrainedComponentsQueueTime[i] +
                constrainedComponents[i].getLookahead();
            targetComponent=constrainedComponents[i];
        }
    }
    result.set(batch, targetComponent);
}
```

Figure 7.5: Earliest Start Time

- Earliest Finish Time

Earliest finish time (EFT) refers to the time that the batch will finish processing if it is scheduled to the batch processing component. Figure 7.6 illustrates the algorithm's pseudo code. The finish time of a batch is calculated by adding the batch's processing time to its start time. The batch's processing time is calculated

by adding the processing time of all baths contained in its recipe.

```
//called by Component Manager when there are new entities in the output queue.
//Component constrainedComponents[];
//EntityQueue outputEntityQueue;
//array stores current time of its constrained component
//double [] constrainedComponentsCurrentTime;
//array stores estimated queue time of constrained components
//double [] constrainedComponentsQueueTime;
float finishTime=Double.max_value;

for (int j=0;j<outputEntityQueue.size;j++){
    Batch batch=(Batch) outputEntityQueue[j];
    for (int i=0;i<constrainedComponents.size;i++){
        if (finishTime>constrainedComponentsCurrentTime[i] +
            constrainedComponentsQueueTime[i] +
            constrainedComponents[i].getLookahead()+
            batch.processTime){
            finishTime=constrainedComponentsCurrentTime[i] +
                constrainedComponentsQueueTime[i] +
                constrainedComponents[i].getLookahead()+
                batch.processTime;
            targetComponent=constrainedComponents[i];
        }
    }
    result.set(batch, targetComponent);
}
```

Figure 7.6: Earliest Finish Time

- Same Recipe:

The same recipe algorithm will send a batch to the batch processing component that is currently processing the same type of batch. However if no match is found, the earliest start time algorithm will be applied. Figure 7.7 illustrates the algorithm's pseudo code.

As these routing rules share the same interface, they can be replaced by each other easily. To do this, the client just needs to change the rule name in the simulation description file and the rule type files.

```

//called by Component Manager when there are new entities in the output queue.
//Component constrainedComponents[];
//EntityQueue outputEntityQueue;
//array stores current recipe of constrained components
//Recipe [] constrainedComponentsCurrentRecipe;
for (int j=0;j<outputEntityQueue.size;j++){
    Batch batch=(Batch) outputEntityQueue[j];
    for (int i=0;i<constrainedComponents.size;i++){
        if (batch.recipe.name==constrainedComponentsCurrentRecipe[i].name){
            targetComponent=constrainedComponents[i];
        }
    }
    result.set(batch, targetComponent);
}

if(result.isEmpty()==true){
    //if no same recipe batch is found,
    //call Earliest Start Time Algorithm here
    EST();
}

```

Figure 7.7: Same Recipe

7.1.2 Batch Processing Component

As mentioned in the last section, the batch processing component receives batches from the wet bench allocation component and processes them through baths. A batch processing component contains a number of different types of baths. There are also a number of predefined recipes stored in the batch processing component.

Figure 7.8, Figure 7.9 and Figure 7.10 demonstrate the implementation of the operation in the batch processing component. The operation is broken down into two parts, i.e. execute and postExecution. In the execute method, after batches are inserted into corresponding baths, the shared variables constrainedComponentsCurrentRecipe and constrainedComponentsQueueTime are updated. These variables are used by the wet bench allocation component for the global dispatching rules. The shared variable manager will push these updates to the wet bench allocation component. Then the minimumFinishTime, which is the minimum finish time among all baths, is calculated and an internal event is scheduled with timestamp minimumFinishTime by inserting it into the internal

event queue. This `minimumFinishTime` indicates the next time to which the operation needs to advance in order to process the batches.

When the time advance request associated with this internal event is granted, the `postExecution` method will be invoked by the time manager. In the `postExecution` method, the `minimumFinishTime` is checked for each bath. If the `minimumFinishTime` is equal to the current time, the batch will either be moved to the next bath or moved to the `outputEntityQueue` if it has finished processing in the wet bench.

To maximize the utilization of baths of the batch processing component, we need to schedule the batches carefully. There are three local dispatching algorithms that we can use, i.e. priority based on FIFO, shortest processing time and same recipe. These are implemented as operation rules of the simulation components.

```
public class WetBenchOperation extends Operation {
    //the baths available in this wet bench
    Bath baths[];
    //the last batch scheduled into the wet bench
    Batch currentBatch;

    //this is a simplified version of checking whether
    //a batch can be scheduled into web bench.
    public boolean ifSchedulable(Batch batch){
        boolean ifFoundBath;
        for(int i=0;i<batch.baths.size;i++){
            ifFoundBath=false;
            for(int j=0;j<baths.size;j++){
                if(batch.recipe.bath[i].name==baths[j].name)
                    if(baths[j].occupied)
                        return false;
                else
                    ifFoundBath=true;
            }
            //not able to find available bath
            if(!ifFoundBath)
                return false;
        }
        return true;
    }
}
```

Figure 7.8: Batch Processing Component Operation Part 1

```

//insert batch into the first bath of its
//recipe and set baths states accordingly
public void insert(Batch batch){
    Recipe recipe;
    recipe=batch.recipe;
    batch.currentStep=0;
    double currentTime=localTimeManager.getCurrentTime();
    for(int i=0;i<baths.size;i++){
        if(baths[i].name==recipe.baths[0].name){
            baths[i].occupied=true;
            baths[i].finishTime=currentTime+baths[i].processTime;
            baths[i].batchInprocess=batch;
        }
    }
}

//check if batch finished processing
public boolean isBatchFinishedProcessing(Batch batch){
    Recipe recipe;
    recipe=batch.recipe;
    if(batch.currentStep==recipe.baths.size)
        return true;
    else
        return false;
}

//execute
public void execute(Hashtable inputEntities) {
    double minimumFinishTime=Double.MAX_VALUE;
    double currentTime=localTimeManager.getCurrentTime();
    //insert batches into baths
    for(int i=0;i<inputEntities.size;i++){
        Batch batch=(Batch) inputEntities[i];
        //this method inserts batch into the first bath of its
        //recipe and set bath's states accordingly
        insert(batch);
        //update shared variables "constrainedComponentsCurrentRecipe"
        //and "constrainedComponentsQueueTime"
        //used in global dispatching rule of Wet Bench Allocation Component
        constrainedComponentsCurrentRecipe=batch.recipe;
        //component queue time is the last entity's actual queue time
        constrainedComponentsQueueTime=currentTime-batch.recvTime;
    }

    //find minimumFinishTime for this round
    for(int i=0;i<baths.size;i++){
        if(baths[i].occupied==true && baths[i].finishTime<minimumFinishTime){
            minimumFinishTime=baths[i].finishTime;
        }
    }

    //insert time advance request event
    Event event=new Event(minimumFinishTime, "time advance request");
    internalEventQueue.insertEvent(event);
}

```

Figure 7.9: Batch Processing Component Operation Part 2

```

//post execution
//it will be called when the time advance request event is executed
public void postExecution(){
    double currentTime=localTimeManager.getCurrentTime();
    //check if a bath has finished processing a batch
    for(int i=baths.size-1;i>=0;i--){
        if(baths[i].finishTime==currentTime){
            currentBatch=baths[i].batchInProcess;
            currentBatch.currentStep++;
            //if batch finished processing, move it to outputEntityQueue.
            if(isBatchFinishedProcessing(currentBatch)){
                outputEntityQueue.insert(currentBatch);
            }
            //else batch still need further processing, move it to the next bath
            else{
                for(int j=0;j<baths.size;j++){
                    if(baths[j].name==currentBatch.recipe.bath[currentBatch.currentStep].name){
                        baths[j].batchInProcess=currentBatch;
                        baths[j].occupied=true;
                    }
                }
                //reset current baths states
                baths[i].batchInProcess=null;
                baths[i].occupied=false;
                baths[i].finishTime=Double.MAX_VALUE;
            }
        }
    }

    //recalculate minimumFinishTime
    for(int i=0;i<baths.size;i++){
        if(baths[i].occupied==true && baths[i].finishTime<minimumFinishTime){
            minimumFinishTime=baths[i].finishTime;
        }
    }

    //insert new time advance request event
    Event event=new Event(minimumFinishTime, "time advance request");
    internalEventQueue.insertEvent(event);
}
} //end of WetBenchOperation class

```

Figure 7.10: Batch Processing Component Operation Part 3

- FIFO:

If the batch processing component uses priority based on FIFO as its operation rule, the batches that can be scheduled into the wet bench will enter the wet bench according to their arrival order. Figure 7.11 gives the pseudo code for the algorithm.

```
//this queue stores entities that are eligible to be processed
//EntityQueue internalEntityQueue;
//this is the wet bench operation
//WetBenchOperation benchOperation;
List resultBatches=new List();
for (int i=0;i<internalEntityQueue.size;i++){
    Batch batch=(Batch) internalEntityQueue[i];
    if(benchOperation.ifSchedulable(batch)){
        resultBatches.add(batch);
    }
}
result.set(resultBatches, benchOperation);
```

Figure 7.11: FIFO

- Shortest Processing Time:

The shortest processing time algorithm will try to schedule the batches into the wet bench according to their processing time where the batch with the shortest processing time has the highest priority. Figure 7.12 gives the pseudo code for the algorithm.

- Same Recipe:

The same recipe algorithm will try to select batches with the same recipe as the current recipe to enter the wet bench. If no batch with the same recipe is found that is schedulable, the FIFO algorithm will be invoked. Figure 7.13 gives the pseudo code for the algorithm.

```

//this queue stores entities that are eligible to be processed
//EntityQueue internalEntityQueue;
//this is the wet bench operation
//WetBenchOperation benchOperation;
List resultBatches=new List();
//if shortestTimeID is -1, there is no batch that can be scheduled
int shortestTimeID=0;
while(shortestTimeID!=-1){
    shortestTimeID=-1;
    double shortestTime=Double.MAX_VALUE;
    for(int i=0;i<internalEntityQueue.size;i++){
        Batch batch=(Batch) internalEntityQueue[i];
        if(benchOperation.ifSchedulable(batch) &&
            batch.processTime<shortestTime &&
            (resultBatches.contains(batch)==false)){
            shortestTime=batch.processTime;
            shortestTimeID=i;
        }
    }
    if(shortestTimeID!=-1)
        resultBatches.add(internalEntityQueue.get(shortestTimeID));
}
result.set(resultBatches, benchOperation);

```

Figure 7.12: Shortest Processing Time

```

//this queue stores entities that are eligible to be processed
//EntityQueue internalEntityQueue;
//this is the wet bench operation
//WetBenchOperation benchOperation;
List resultBatches=new List();
for(int i=0;i<internalEntityQueue.size;i++){
    Batch batch=(Batch) internalEntityQueue[i];
    if(benchOperation.ifSchedulable(batch)&&
        benchOperation.currentRecipe.name==batch.recipe.name){
        resultBatches.add(batch);
    }
}
result.set(resultBatches,benchOperation);
//when there is no same recipe batch can be found
if(resultBatches.isEmpty()==true)
    FIFO();

```

Figure 7.13: Same Recipe

7.2 Objectives of the Case Study

The first objective of the case study is to find the combination of global dispatching algorithm and local dispatching algorithm that gives us the shortest cycle time and waiting

Table 7.1: Baths associated with each Wet Bench

Wet Bench	Bath													
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	X	X			X	X	X	X				X	X	X
2	X	X			X	X	X	X				X	X	X
3	X	X			X	X	X	X			X	X	X	X
4	X	X			X	X	X	X			X	X	X	X
5	X	X	X	X	X	X	X	X	X		X	X	X	X
6	X	X			X	X	X	X	X	X		X	X	X
7					X	X	X	X				X	X	X
8					X	X	X	X				X	X	X

time. After that, results from the experiment will be analyzed to explore the possibility to improve the performance by changing the recipes that a wet bench can process. The modifications do not require changing the wet benches' hardware, only to add or remove recipes. We can then evaluate the modification by repeating the experiment and comparing it to the original performance.

7.2.1 Experiment Design

The simulation is conducted on a cluster with ten nodes. There is one batch generating component in the system and eight batch processing components. The batch generating component randomly generates one of 34 different recipes for batches using an inter-arrival time with a mean uniform distribution of 180. There are three available global dispatching algorithms, i.e. earliest start time (ESF), earliest finish time (EFT) and same recipe (SR). These algorithms are implemented as routing rules in the batch generating component.

The eight batch processing components represent different types of wet benches with baths as shown in Table 7.1. With different baths available in different wet benches, they

are capable of processing different types of recipes. Table 7.2 shows all possible recipes that different wet benches are capable of processing. The set of recipes that each wet bench is originally configured to process is shown in Table 7.3.

There are three local dispatching algorithms available for each batch processing component, i.e. FIFO, same recipe (SR) and shortest processing time (SPT). These rules are implemented as operation rules inside a batch processing component. With different combinations of global dispatching algorithm and local dispatching algorithm, we conduct experiments based on nine possible combinations, i.e. EST-FIFO, EFT-FIFO, SR-FIFO, EST-SR, EFT-SR, SR-SR, EST-SPT, EFT-SPT AND SR-SPT. The batch generator generates 2000 batches in each simulation run and each combination is repeated five times.

7.2.2 Experimental Results

The results for the nine combinations of global and local dispatching algorithms are shown in Figure 7.14, Figure 7.15 and Table 7.4.

The overall average waiting time and the overall average cycle time are calculated by averaging the average waiting time and the average cycle time respectively over the simulation runs for different combinations. We observe that EST-SR gives the shortest average waiting time 966 and EFT-SPT gives the shortest average cycle time 3798. This is because the routing rule EST focuses on the earliest time that a batch can be scheduled into the wet bench, but both the routing rule EFT and the operation rule SPT focus on the shortest time in which a batch can finish processing. Usually cycle time is the most important, and we can choose EFT and SPT as the global dispatching algorithm and local dispatching algorithm respectively with the current configuration of wet benches.

Table 7.2: Possible Recipes handled by each Wet Bench

	Wet Bench							
Recipe ID	1	2	3	4	5	6	7	8
R1	X	X	X	X	X	X	X	X
R2	X	X	X	X	X	X	X	X
R3	X	X	X	X	X	X	X	X
R4	X	X	X	X	X	X	X	X
R5	X	X	X	X	X	X	X	X
R6	X	X	X	X	X	X	X	X
R7	X	X	X	X	X	X		
R8	X	X	X	X	X	X		
R9	X	X	X	X	X	X		
R10	X	X	X	X	X	X		
R11	X	X	X	X	X	X		
R12	X	X	X	X	X	X		
R13	X	X	X	X	X	X		
R14	X	X	X	X	X	X		
R15	X	X	X	X	X	X		
R16	X	X	X	X	X	X	X	X
R17	X	X	X	X	X	X		
R18			X	X	X		X	X
R19			X	X	X		X	X
R20			X	X	X		X	X
R21			X	X	X		X	X
R22			X	X	X		X	X
R23			X	X	X			
R24			X	X	X			
R25			X	X	X			
R26	X	X	X	X	X	X		
R27	X	X	X	X	X	X	X	X
R28			X	X	X			
R29	X	X	X	X	X	X		
R30			X	X	X			
R31			X	X	X			
R32			X	X	X			
R33	X	X	X	X	X	X	X	X
R34			X	X	X			

Table 7.3: Original Recipes handled by each Wet Bench

Recipe ID	Wet Bench							
	1	2	3	4	5	6	7	8
R1		X			X	X	X	X
R2		X		X	X	X	X	X
R3		X		X	X	X	X	X
R4	X	X	X	X	X	X	X	X
R5	X	X	X	X	X		X	X
R6	X	X	X	X	X		X	X
R7		X			X			
R8		X			X			
R9		X		X	X			
R10		X		X	X			
R11		X		X	X			
R12	X	X	X	X				
R13	X		X					
R14	X	X	X	X	X			
R15	X		X					
R16	X	X	X	X	X			
R17	X	X	X	X	X			
R18				X	X			
R19				X	X			
R20					X			
R21				X	X			
R22				X	X			
R23				X	X			
R24				X	X			
R25				X	X			
R26			X	X				
R27				X	X			
R28				X	X			
R29				X	X			
R30				X	X			
R31				X	X			
R32				X	X			
R33				X	X			
R34				X	X

Figure 7.14: Average Waiting Time in Seconds

Figure 7.15: Average Cycle Time in Seconds

To improve the performance of the wet benches, we have to look at the performance of individual benches to find if there are any bottleneck benches in the system.

Table 7.4: Experimental Results in Seconds with Original Recipes

Rules	Average Waiting Time	Average Cycle Time
EST-FIFO	1016	3952
EFT-FIFO	1007	3905
SR-FIFO	1061	4066
EST-SR	966	3856
EFT-SR	1128	4195
SR-SR	1128	4208
EST-SPT	982	3858
EFT-SPT	1006	3798
SR-SPT	1250	4426
Overall Average	1060	4029

Table 7.5: Average Waiting Time in Seconds with Original Recipes

Rules	WB1	WB2	WB3	WB4	WB5	WB6	WB7	WB8
EST-FIFO	405.6	1150.4	344.6	2883.2	2836	173.8	125	211.4
EFT-FIFO	336.8	1245	308.6	2740.2	2941.2	140.4	110.2	238
SR-FIFO	864	1406	404.4	2924.8	2535.8	226.4	128.2	0
EST-SR	361.8	1107.4	325.6	2730.6	2709.2	145.6	123.4	225.4
EFT-SR	376	1143.8	319.8	3361.4	3363	120.4	127	214.6
SR-SR	835.8	1461.6	422.2	3188	2789.2	207.6	124	0
EST-SPT	486	1223.4	336.4	2652.8	2615	162.8	197.4	186.4
EFT-SPT	375.8	1151	314.6	2686	2802.8	143.2	155	422.2
SR-SPT	898	1419	439.4	3587.2	3265.8	250.2	118.2	27.4
Weighting Percentage	6%	15%	4%	35%	30%	3%	2%	2%

Table 7.5 and Table 7.6 display the average waiting time and the average cycle time for each wet bench when we conduct simulations with different rule combinations. From the data, we observe that wet benches 4 and 5 have a much longer waiting time and cycle time than other wet benches in the system. The weighting percentage for each wet

Table 7.6: Average Cycle Time in Seconds with Original Recipes

Rules	WB1	WB2	WB3	WB4	WB5	WB6	WB7	WB8
EST-FIFO	2535.8	3264.8	2698	5268.2	5070	1763.4	1990.2	2157
EFT-FIFO	2792.6	3351	2673.2	5109.4	4960.2	1828.2	2001.2	2184.8
SR-FIFO	3327	3509.8	2807	5309.6	4758.4	1834.2	1871.2	1836
EST-SR	2771.2	3211.8	2655.6	5092.2	4916.8	1810.2	2024.6	2200.8
EFT-SR	2779	3254.8	2707.4	5735.4	5581	1818.4	1988	2176
SR-SR	3277	3533	2804.8	5569.8	5010	1874.6	1862.2	1620
EST-SPT	2892.2	3328.4	2780.6	5041	4843	1821.8	1988.6	2014.8
EFT-SPT	2789.6	3259.6	2723.6	4663	5041.8	1837.8	2038.4	2197.2
SR-SPT	3372	3498	2911.2	6044	5497.2	1821.6	1833.6	1670.2
Weighting Percentage	12%	13%	11%	21%	20%	7%	8%	0.08%

bench is calculated by dividing the average waiting time/average cycle time of that wet bench by the sum of the average waiting time/average cycle times of all wet benches. Wet benches 4 and 5 show a weighting percentage of 35% and 30% in Table 7.5 and 21% and 20% in Table 7.6, thus we conclude that wet benches 4 and 5 are potential bottleneck wet benches. If we can distribute some load to other wet benches, the performance of the whole system may show significant improvement. After analyzing the recipes for wet benches 4 and 5, we find that only wet benches 4 and 5 are configured to process recipes 18-25 and recipes 27-34. In other words, 47% of the recipes can only be processed by these two wet benches. This is the reason why these two wet benches have a much longer waiting time and cycle time than others. In addition, wet bench 4 is also configured to process recipes 2-6, 9-12, 14, 16, 17 and wet bench 5 is configured to process recipes 1-11, 14 and 16. Although these recipes can also be processed by other wet benches, it still increases the load of wet benches 4 and 5. By analyzing all the possible recipes handled by each wet bench (see Table 7.2), we decide to adjust the recipes that are handled by wet benches 4 and 5 and also some other wet benches as shown in Table 7.7. Recipes 2-6, 9-12, 14, 16, 17, 27, 28 and 33 are now removed from wet bench 4, recipes 1-7, 27, 28 and

Table 7.7: Modified Recipes handled by each Wet Bench

	Wet Bench							
Recipe ID	1	2	3	4	5	6	7	8
R1		X				X	X	X
R2		X				X	X	X
R3		X				X	X	X
R4	X	X	X			X	X	X
R5	X	X	X				X	X
R6	X	X	X				X	X
R7		X						
R8		X			X			
R9		X			X			
R10		X			X			
R11		X			X			
R12	X	X	X					
R13	X		X					
R14	X	X	X		X			
R15	X		X					
R16	X	X	X		X			
R17	X	X	X		X			
R18				X	X			
R19				X	X			
R20					X			
R21				X	X			
R22				X	X			
R23				X	X			
R24				X	X			
R25				X	X			
R26			X	X				
R27						X	X	X
R28						X	X	X
R29				X	X			
R30				X	X			
R31				X	X			
R32				X	X			
R33						X	X	X
R34				X	X			

33 are removed from wet bench 5. Meanwhile, recipes 27, 28 and 33 are added to wet benches 6, 7 and 8. The objective of this modification is to balance the load between all wet benches. The results obtained with the modified wet bench configuration are shown in Table 7.8, Figure 7.16 and Figure 7.17.

Table 7.8: Experimental Results in Seconds with Modified Recipes

Rules	Average Waiting Time	Average Cycle Time
EST-FIFO	564	2966
EFT-FIFO	551	2897
SR-FIFO	704	3237
EST-SR	574	3002
EFT-SR	597	2934
SR-SR	743	3296
EST-SPT	574	3001
EFT-SPT	562	2985
SR-SPT	683	3205
Overall Average	617	3058

From Table 7.8, Figure 7.16 and Figure 7.17, we see that the modified wet bench configuration has much better performance than the original one. The overall average waiting time has decreased by 41.8% from 1060 to 617 and the overall average cycle time has decreased by 24.1% from 4029 to 3058.

Comparing the average waiting time and average cycle time for each wet bench in Table 7.9 and Table 7.10 to Table 7.5 and Table 7.6, we can see that after altering the wet bench configuration, the wet bench system has much better load balancing. The weighting percentage for average waiting time for wet benches 4 and 5 has decreased from 35% and 30% to 24% and 24% respectively and that for the average cycle time has decreased from 21% and 21% to 17% and 15% respectively. This effectively increases the total performance of the system. From Table 7.10, we find that the combination EFT-FIFO has the shortest average cycle time, 2897. However, in the original configuration,

Figure 7.16: Average Waiting Time in Seconds with Modified Recipes

Table 7.9: Average Waiting Time in Seconds with Modified Recipes

Rules	WB1	WB2	WB3	WB4	WB5	WB6	WB7	WB8
EST-FIFO	359.8	977	318.2	1043.8	1166.2	176.4	189.6	283.6
EFT-FIFO	331.2	941.2	285.2	1020.4	1154.2	188.8	196.2	295.2
SR-FIFO	934.2	1310.2	413.2	1295.8	1117.2	393.2	121.2	54.8
EST-SR	374.4	959.8	297	1104.2	1197.8	189.2	175.2	298.8
EFT-SR	614.8	972.2	333	1060.4	1162	187.6	173.6	279.4
SR-SR	861.6	1279.4	423.6	1486.4	1283.6	391.6	127.6	93
EST-SPT	364.8	925.6	327.2	1130.8	1233	172.8	163	281.4
EFT-SPT	356.6	930	302.4	1055.2	1195.8	179.2	187.6	289.6
SR-SPT	879.8	1240.4	374.6	1314.8	1075.2	395.8	99	89.6
Weighting Percentage	11%	22%	7%	24%	24%	5%	3%	4%

Figure 7.17: Average Cycle Time in Seconds with Modified Recipes

Table 7.10: Average Cycle Time in Seconds with Modified Recipes

Rules	WB1	WB2	WB3	WB4	WB5	WB6	WB7	WB8
EST-FIFO	2720.4	3056.6	2701.6	3507.2	3455.8	1961.4	2092.6	2278.2
EFT-FIFO	2717.2	3033.6	2673.8	3290.8	3421.8	1975	2101	2271.8
SR-FIFO	3359.8	3387.4	2836.2	3786.6	3383	2213.4	1789.6	1728.2
EST-SR	2789.4	3065.6	2756.8	3587.6	3486.2	1965.4	2085.8	2299.2
EFT-SR	2247.4	3075.4	2713.4	3532.6	3450.2	1962.2	2072.6	2276.4
SR-SR	3345.2	3359.8	2854	3943.4	3566.2	2237.2	1797.6	1626.4
EST-SPT	2801.6	3013	2682.4	3621.2	3523.6	1973	2058.4	2297.6
EFT-SPT	2855.8	3023.4	2708.2	3538.8	3473.4	1985.8	2091.4	2285.4
SR-SPT	3327.4	3320.4	2776.8	3812	3331	2199	1730.8	1987
Weighting Percentage	13%	14%	12%	17%	15%	10%	9%	10%

EFT-SPT gives the shortest cycle time. The reason is that with the previous setup, the recipe distribution is not balanced. Some recipes can only be processed by two wet benches out of eight. A local dispatching rule like SPT which focuses on reducing individual processing times can improve the performance of the wet bench. However, after the recipes for the wet benches are rearranged, FIFO has better performance than SPT. This is because it keeps the choice made by the global dispatching rule EFT, which focuses on minimizing the overall cycle time. Under the current configuration, EFT-FIFO should be chosen as dispatching rule combination.

7.3 Summary

In this chapter, a case study on the use of the rule engine in the simulation of multi-recipes wet benches in the wafer fabrication process is presented. Global dispatching algorithms and local dispatching algorithms are implemented as routing rules and operation rules respectively. With the help of the rule engine, we can replace different rules easily by providing different rule files to the simulation component. As we can conveniently switch between different rules, the study of the performance of different combinations of global dispatching algorithm and local dispatching algorithm can be carried out smoothly. After analyzing different combinations, we find the combinations with the shortest waiting time and cycle time. We expect these to give the best performance under the original system configuration. In the next step, we study the configuration of wet benches and make changes to the recipes of certain wet benches. We find that the changes significantly improve the system performance, i.e. a 41.8% decrease in average waiting time and a 24.1% decrease in average cycle time. In addition, we find that different configurations of wet benches change the performance of the rule combinations. Now the combination EFT-FIFO gives the shortest cycle time whereas the combination EFT-SPT has the best performance with original configuration. To summarize, from this case study, we have

shown that the local rule engine is capable of implementing dispatching algorithms in a real scenario. In addition, it is also easy to switch between different global and local dispatching algorithms by providing the corresponding rule files and changing the settings in the simulation description file.

Chapter 8

Conclusions and Future Work

8.1 Achievements

A Framework for Component-based Distributed Simulation on the Grid (SOAr-DSGrid) was successfully designed and developed in this research project.

The framework is based on a Service Oriented Architecture (SOA). This enables the simulation developers to separate the implementation and interface of services. There are four types of services available in the framework, i.e. simulation component, organizer, index service and repository. These services are implemented as Grid services. A simulation component is designed to simulate particular objects by a service provider. The organizer manages simulation components. It can start, monitor and terminate their execution. A client contacts the organizer to conduct a simulation session. The index service stores the address of other services. It is a well-known service-discovery service used by all clients in the framework. The repository is used to store five kinds of files, namely component interface description files, composite component implementation files, entity type files, rule type files and rule files. With the Service-Oriented Architecture, forming and executing a simulation session becomes very simple for a client.

The development of simulation components in the framework adopts a component-

based approach. The most important characteristic of the component-based approach is reusability. In the framework, a simulation component designed for one simulation system can be reused in another system. For instance, a radar simulation component that is used to monitor cars in a traffic simulation session can be reused in another simulation session to monitor ships. This approach helps to reduce the development cost for new simulation systems if existing simulation components can be reused.

Another important objective of the framework is for it to be distributed. A distributed system usually provides better scalability than a centralized system. To achieve fully distributed simulation execution, there is no centralized service during runtime. In the framework, communication between different simulation components occurs when they want to advance logical time or to send data. Each simulation component has a local time manager and local rule engine to enable distributed data flow and control flow.

Time management is always a critical part in any distributed simulation system. It ensures temporal causality among simulated events. If the time manager is not properly designed, it will severely affect the performance of the whole simulation system. The objective of the time manager in this project is to enable the distributed computation of logical time. A local time manager was developed and a Next Event Request (NER) algorithm was implemented in the time manager. A component can simply issue a NER when it wants to advance its logical time. As there is no centralized time manager, we avoid the potential bottleneck problem when many components send a time advance request to the centralized time manager at same time causing it to be slow to respond. A performance evaluation is presented in Chapter 6 for the distributed time manager. Two models are built for the experiments, i.e. the Super-Ping model and the N-way Connection model. The Super-Ping model is used to examine the scalability of the distributed time manager and the N-way Connection model is used to examine the effect of connectivity on the distributed time manager. The results of the Super-Ping model experiments show that

the distributed time manager has better scalability than a centralized time manager. The N-way Connection experiments indicate that when the connections between simulation components are sparse, the distributed time manager performs better.

The rule engine enables data flow and control flow to be distributed in our framework. There are two types of rules: operation rules and routing rules. Operation rules are used to select the appropriate operation to process entities and routing rules are used to select the appropriate component to send entities. Rules are written in Java syntax and interpreted by BeanShell. Rules can be configured in rule files. In Chapter 7, a case study of using the rule engine to implement global and local dispatching rules in multi-recipes wet benches is presented. The case study shows that the rule engine is capable of executing different rules and that rules can be swapped easily by providing different rule configuration files.

8.2 Future Work

Although a lot of effort has been put into the project, there are still some areas that can be further studied.

The mechanism for creating composite simulation components by combining primitive components can also be implemented. With composite simulation components, simulation developers can create new simulation components by reusing existing primitive components instead of creating components from scratch. This can reduce the development cost of constructing a new simulation system.

Currently, it is assumed that only one operation can be active at one time. In future, the component manager can be implemented to support execution of multiple operations simultaneously. This will enable simulation components to simulate more complex objects.

Regarding time management, currently only the Next Event Request algorithm is implemented. If we implement the Time Advance Request algorithm [32], other types of simulation system will be supported by the framework, i.e. time-stepped simulation systems. In addition, the current Next Event Request algorithm assumes that the lookahead from one component to another is always greater than zero^{8.1}. However, prohibiting zero lookahead can be restrictive when we design a simulation system. In the future, we can improve our time algorithm to support zero lookahead.

Unlike the component manager which is able to switch between different operations during runtime, the local rule engine can only execute one operation rule and one routing rule during runtime. If the local rule engine can dynamically load rules, it will give more flexibility to simulation developers when they design simulation components and rules.

^{8.1}Strictly, there must be no cycle of simulation components where the lookahead between each pair of neighboring components in the cycle is zero.

Bibliography

- [1] IEEE Standard 1516 (HLA Rules), 1516.1 (Interface Specification) and 1516.2 (OMT). Sep 2000. <http://www.dmsi.mil/public/transition/hla/>.
- [2] P. Alee and S. Frost. *Component-Based Development for Enterprise Systems: Applying the SELECT Perspective*. Cambridge University Press, New York, NY, USA, 1998.
- [3] T. Altiok and B. Melamed. *SIMULATION MODELING AND ANALYSIS WITH ARENA*. McGraw Hill, Boston, MA , USA, 2000.
- [4] R. G. Bartholet, D. C. Brogan, P. F. Reynolds, and J. C. Carnahan. In Search of the Philosopher’s Stone: Simulation Composability Versus Component-Based Software Design. In *Proceedings of the 2004 Fall SIW*, pages 14 – 28, Sep, 2004.
- [5] H. A. Buss. Component-based Simulation Modeling. In *the 2000 Winter Simulation Conference*, pages 243 – 249, 2000.
- [6] W. T. Cai, S. J. Turner, and H. Zhao. A Load Management System for Running HLA-based Distributed Simulations over the Grid. In *the 6th IEEE International Symposium on Distributed Simulation and Real Time Application*, pages 7 – 14, Oct. 2002.

-
- [7] C. D. Carothers and R. M. Fujimoto. Design and Implementation of HLA Time Management in the RTI Version F.0. In *Winter Simulation Conference*, pages 373 – 380, Dec 1997.
 - [8] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. March, 2001. <http://www.w3.org/TR/wsdl>.
 - [9] OASIS Web Services Resource Framework (WSRF) Technical Committee. OASIS Web Services Resource Framework (WSRF) TC. Jun, 2007. <http://www.oasisopen.org/committees/tchome.php?wgabbrevwsrf>.
 - [10] J. S. Dahmann and K. L. Morse. High Level Architecture for Simulation: An Update. In *the 2nd IEEE International Workshop on Distributed Interactive Simulation and Real-Time Applications*, pages 32 – 40, 1998.
 - [11] P. K. Davis and R. H. Anderson. Improving the Composability of DoD Models and Simulations. In *JDMS: The Journal of Defense MS: Applications, Methodology, Technology*, page Vol1, Jan, 2004.
 - [12] Defense Modeling Department of Defence and Simulation Office. Rti 1.3-next generation programmer’s guide, version 5. 2002.
 - [13] L. Dutka and J. Kitowski. Application of Component-Expert Technology for Selection of Data-handlers in CriossGrid. In *PVM/MPI*, pages 25 – 32, 2002.
 - [14] L. Dutka and J. Kitowski. Flexible Component Architecture for Information WEB Portals. In *International Conference on Computational Science*, pages 629 – 638, 2003.
 - [15] P. Eweka-Esiet and D. Mehandjiska-Stavreva. Why a Component-based Architecture for E-business? In *Computer Science and Technology*, page 394, May 2003.
-

-
- [16] M. Farshad. Component-Based Simulation Model Development Using BOMs and Web Services. In *Modelling and Simulation, 2007. AMS '07. First Asia International Conference*, pages 238 – 246, March, 2007.
 - [17] The Common Component Architecture forum. Terms and Definitions of Common Component Architecture. Jan, 2005. <http://www.cca-forum.org/glossary/index.html>.
 - [18] I. Foster. What is the grid? a three point checklist. July 20, 2002. <http://www-fp.mcs.anl.gov/foster/Articles/WhatIsTheGrid.pdf>.
 - [19] I. Foster and C. Kesselman. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. In *Cluster Computing and the Grid*, pages 6 – 7, 2001.
 - [20] I. Foster, H. Kishimoto, A. Savva, D. Berry, and etc. The Open Grid Services Architecture, Version 1.0. June, 2005. <http://www.gridforum.org/documents/GWD-IE/GFD-I.030.pdf>.
 - [21] R. M. Fujimoto. *Parallel and Distributed Simulations Systems*. Wiley, NJ, 2000.
 - [22] B.P. Gan, M.Y.H. Low, J.H. Wei, X.G. Wang, S.J. Turner, and W.T. Cai. Synchronization and management of shared state in HLA-based distributed simulation. In *Simulation Conference*, pages 847 – 854, Dec 2003.
 - [23] Globus. GT4, Globus Toolkit Version 4.0. 2006. <http://www.globus.org/toolkit/>.
 - [24] Globus. Globus Toolkit 4 Release. 2007. <http://www.nesc.ac.uk/news/articlesandcalls/30042005.html>.
 - [25] P. Herzum and O. Sims. *Business Component Factory: A Comprehensive Overview of Component-Based Development for the Enterprise*. John Wiley & Sons, Inc, Indianapolis, USA, 1999.
-

-
- [26] IBM, BEA Systems, Microsoft, SAP AG, and Siebel Systems. Business Process Execution Language for Web Services version 1.1. Feb, 2007. <http://www.ibm.com/developerworks/library/specification/wsbpel>.
- [27] Unger J. and Haynos M. A visual tour of Open Grid Services Architecture. 2007. <http://www.ibm.com/developerworks/grid/library/gr-visual/>.
- [28] D. Jefferson. Virtual time. In *ICPP*, pages 384–394, 1983.
- [29] R. R. Kodali. What is service-oriented architecture? An introduction to SOA. June, 2005. <http://www.javaworld.com/javaworld/jw-06-2005/jw-0613-soa.html>.
- [30] S. Krishnan and D. Gannon. XCAT3: A Framework for CCA Components as OGSA Services. In *HIPS 2004, 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 90–97, April 2004.
- [31] S. Krishnan, P. Wagstrom, and G. V. Laszewski. GSFL: A Workflow Framework for Grid Services. 19 July, 2002. <http://www.globus.org/cog/papers/gsfl-paper.pdf>.
- [32] F. Kuhl, J. S. Dahmann, and R. Weatherly. *Creating Computer Simulation Systems: An Introduction to the High Level Architecture*. Prentice Hall, Indiana, USA, Oct 8, 1999.
- [33] Jet Propulsion Laboratory. OODT Grid Service – OODT Data Grid Framework. 2005. <http://oodt.jpl.nasa.gov/grid/>.
- [34] B. Lawrence. NERC DataGrid. 2004. <http://ndg.badc.rl.ac.uk/>.
- [35] F. Leymann. Web Service Flow Language (WSFL 1.0). May 2001. <http://www-306.ibm.com/software/solutions/webservices/pdf/WSFL.pdf>.

-
- [36] J. Luthi and S. Grossmann. The Resource Sharing System: Dynamic Federate Mapping for HLA-based Distributed Simulation. In *15th Workshop on Parallel and Distributed Simulation*, pages 91–98, May 2001.
 - [37] K. L. Morse, D. L. Drake, and R. P. Z. Brunton. Web Enabling an Rti - an XMSF Profile. In *the IEEE 2003 European Simulation Interoperability Workshop*, 2003.
 - [38] P. Niemeyer. What is BeanShell? 2007. <http://www.beanshell.org/intro.html>.
 - [39] OASIS. Universal Description, Discovery and Integration. 2006. <http://www.uddi.org/specification.html>.
 - [40] K. Pearson. What is Distributed Computing? August, 2007. <http://distributedcomputing.info/>.
 - [41] R.F. Shannon. *Simulation: A Survey with Research Suggestions*, volume 7. Sept, 1975.
 - [42] J. Snell. The Web services insider, Part 4: Introducing the Web Services Flow Language. Jun, 2001. <http://www.ibm.com/developerworks/webservices/library/ws=ref4/?dwzone=webservices>.
 - [43] IEEE Computer Society. *IEEE Recommended Practice for High Level Architecture (HLA) Federation Development and Execution Process (FEDEP)*. 2003.
 - [44] B. Sotomayor. The Globus Toolkit 4 Programmer’s Tutorial. 2005. <http://gdp.globus.org/gt4-tutorial/multiplehtml/ch01s04.html>.
 - [45] C. M. Sperberg-McQueen and H. Thompson. XML Schema. April, 2000. <http://www.w3.org/XML/Schema>.
 - [46] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, USA, 1998.
-

-
- [47] S. Tuecke, K. Czajkowski, I. Foster, and etc. Open Grid Services Infrastructure (OGSI) Version 1.0. Jan 2005. <http://www-unix.globus.org/toolkit/draft-ggf-ogsi-gridservice-33-2003-06-27.pdf>.
 - [48] W3C. XML Schema, W3C XML Schema. Jan 2005. <http://www.w3.org/XML/Schema>.
 - [49] E. W. Weisel, M. D. Petty, and R. R. Mielke. A Survey of Engineering Approaches to Composability. In *Proceedings of the Spring 2004 SIW*, pages 18 – 23, April, 2004.
 - [50] E. W. Weisel, M. D. Petty, and R. R. Mielke. Validity of Models and Classes of Models in Semantic Composability. In *Proceedings of the Fall 2003 SIW*, pages 14–19, Sept, 2003.
 - [51] P. Wonnacott and D. Bruce. A Prototype Implementation of APOSTLE, and its Performance. In *SCS Summer Computer Simulation Conference*, pages 197–205, 1995.
 - [52] XCAT. The XCAT Project. Jan 2005. <http://www.extreme.indiana.edu/xcat/>.
 - [53] K. Zajac, M. Bubak, M. Malawski, and P. Slood. Towards a Grid Management Systems for HLA-based Interactive Simulations. In *the 7th IEEE International Symposium on Distributed Simulation and Real-Time Applications*, pages 4–11, Oct 2003.
 - [54] K. Zajac, A. Tirado-Ramos, Z. Zhao, and P. Slood. Grid Services for HLA-based Distributed Simulation Frameworks. In *1st European AcrossGrids Conference*, pages 147–154, Feb 2003.
 - [55] T. Zhang, C. Zhang, Y. Liu, and Y. Zha. A Design of Distributed Simulation Based on GT3 Core. In *Grid and Cooperative Computing: Second International Workshop*, pages 590–596, Feb 2003.
-

- [56] S.H. Zhu, Z.H. Du, and X.D. Chai. GDSA: A Grid-Based Distributed Simulation Architecture. In *Cluster Computing and the Grid Workshops, 2006. Sixth IEEE International Symposium on*, pages 66 – 66, May 2006.

Appendices

Appendix A

Component Interface Description File

```
<?xml version="1.0" encoding="UTF-8"?> <!-- edited with XMLSpy
v2005 U (http://www.xmlspy.com) by Xinjun Chen (NTU) -->
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:comp="http://www.cbdsg.org/comp"
targetNamespace="http://www.cbdsg.org/comp"
elementFormDefault="unqualified"
attributeFormDefault="unqualified">
  <xsd:annotation>
    <xsd:documentation>
      Interface Schema
      2004 PDCC NTU Singapore.
    </xsd:documentation>
  </xsd:annotation>
  <!-- root element -->
  <xsd:element name="component" type="comp:componentType"/>
  <!-- Definition of global simpleType "versionType" -->
  <xsd:simpleType name="versionType">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="[0-9]+\.[0-9]+"/>
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:complexType name="componentType">
    <xsd:sequence>
      <!-- Component description -->
      <xsd:element name="description" type="xsd:string"/>
      <xsd:element name="operations">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="operation" maxOccurs="unbounded">
              <xsd:complexType>
                <xsd:sequence>
                  <xsd:element name="name" type="xsd:string"/>
                  <!-- operation description -->
                  <xsd:element name="description" type="xsd:string"/>
                </xsd:sequence>
              </xsd:complexType>
            </xsd:element>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

```

        <xsd:element name="inputEntity">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element name="type" type="xsd:string"
                        maxOccurs="unbounded"/>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>
        <xsd:element name="outputEntity">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element name="type" type="xsd:string"
                        maxOccurs="unbounded"/>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>
    </xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
<xsd:element name="variables">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="variable" type="comp:attributeType"
                maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
</xsd:sequence>
<xsd:attribute name="type" type="xsd:string" use="required"/>
<xsd:attribute name="version" type="comp:versionType" use="required"/>
<xsd:attribute name="isComposite" type="xsd:boolean" use="required"/>
<xsd:attribute name="provider" type="xsd:string" use="required"/>
</xsd:complexType>
<xsd:complexType name="attributeType">
    <xsd:sequence>
        <xsd:element name="name" type="xsd:string"/>
        <xsd:element name="type" type="xsd:string"/>
        <!-- Description of variable -->
        <xsd:element name="description" type="xsd:string"/>
        <xsd:element name="defaultValue" type="xsd:string" minOccurs="0"/>
        <xsd:element name="sharable" type="xsd:boolean"/>
    </xsd:sequence>
</xsd:complexType>
</xsd:schema>

```

Appendix B

Composite Component Implementation File

```
<?xml version="1.0" encoding="utf-8"?> <!-- edited with XMLSpy
v2005 U (http://www.xmlspy.com) by Xinjun Chen (NTU) -->
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:impl="http://www.ccm.org/impl"
targetNamespace="http://www.ccm.org/impl">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Component Implementation Schema
      2004 PDCC NTU Singapore.
    </xsd:documentation>
  </xsd:annotation>
  <!-- Root element -->
  <xsd:element name="compositeComponentImpl" type="impl:compositeComponentImplType"/>
  <xsd:complexType name="compositeComponentImplType">
    <xsd:sequence>
      <!-- constituent components involved in the composite
      component from the view point of the client -->
      <xsd:element name="componentInstances">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="componentInstance"
              type="impl:componentInstanceType" maxOccurs="unbounded"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="composition">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="entityFlows">
              <xsd:complexType>
                <xsd:sequence>
                  <xsd:element name="entityFlow"
                    type="impl:entityFlowType" maxOccurs="unbounded"/>
                </xsd:sequence>
              </xsd:complexType>
            </xsd:element>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

```

        <xsd:element name="variableFlows">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element name="variableFlow"
                        type="impl:variableFlowType" maxOccurs="unbounded"/>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>
    </xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:sequence>
<xsd:attribute name="type" type="xsd:string" use="required"/>
<xsd:attribute name="version" type="impl:versionType" use="required"/>
<xsd:attribute name="provider" type="xsd:string" use="required"/>
</xsd:complexType>
<xsd:complexType name="componentInstanceType">
    <xsd:sequence>
        <xsd:element name="setupVariable" maxOccurs="unbounded">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element name="name" type="xsd:string"/>
                    <xsd:element name="type" type="xsd:string"/>
                    <xsd:element name="value" type="xsd:string"/>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>
        <xsd:element name="operationRule" type="impl:ruleConfiguration"
            minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="type" type="xsd:string" use="required"/>
    <xsd:attribute name="version" type="xsd:string" use="required"/>
    <xsd:attribute name="isComposite" type="xsd:boolean" use="required"/>
</xsd:complexType>
<xsd:complexType name="entityFlowType">
    <xsd:sequence>
        <xsd:element name="entryComponent" type="xsd:string"/>
        <xsd:element name="entityLink" maxOccurs="unbounded">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element name="source" type="xsd:string"/>
                    <xsd:element name="targets">
                        <xsd:complexType>
                            <xsd:sequence>
                                <xsd:element name="target"
                                    type="xsd:string" maxOccurs="unbounded"/>
                            </xsd:sequence>
                        </xsd:complexType>
                    </xsd:element>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>
        <xsd:element name="routingRule"

```

```

        type="impl:ruleConfiguration" minOccurs="0"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:sequence>
<xsd:attribute name="entityType" type="xsd:string" use="required"/>
</xsd:complexType>
<xsd:complexType name="variableFlowType">
  <xsd:sequence>
    <xsd:element name="provider" type="xsd:string"/>
    <xsd:element name="subscribers">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="subscriber" maxOccurs="unbounded"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string" use="required"/>
</xsd:complexType>
<xsd:simpleType name="versionType">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="[0-9]+\.[0-9]+"/>
  </xsd:restriction>
</xsd:simpleType>
<xsd:complexType name="ruleConfiguration">
  <xsd:sequence>
    <xsd:element name="configVar"
      type="impl:configurationVariableType" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string" use="required"/>
  <xsd:attribute name="type" type="xsd:string" use="required"/>
  <xsd:attribute name="uri" type="xsd:string"/>
  <xsd:attribute name="applyToEntity" type="xsd:string" use="required"/>
</xsd:complexType>
<xsd:complexType name="configurationVariableType">
  <xsd:choice>
    <xsd:sequence>
      <xsd:element name="type" type="xsd:string"/>
      <xsd:element name="value" type="xsd:string"/>
    </xsd:sequence>
    <xsd:sequence>
      <xsd:element name="objectName" type="xsd:string"/>
      <xsd:element name="variableName" type="xsd:string"/>
    </xsd:sequence>
  </xsd:choice>
  <xsd:attribute name="name" type="xsd:string" use="required"/>
</xsd:complexType>
</xsd:schema>

```

Appendix C

Simulation Description File

```
<?xml version="1.0" encoding="utf-8"?> <xsd:schema
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:sim="http://www.ccm.org/sim"
targetNamespace="http://www.ccm.org/sim"
elementFormDefault="unqualified"
attributeFormDefault="unqualified">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Description Schema
      2004 PDCC NTU Singapore.
    </xsd:documentation>
  </xsd:annotation>
  <!-- The root element -->
  <xsd:element name="simulation" type="sim:simulationType"/>
  <xsd:attributeGroup name="componentAttributes">
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="type" type="xsd:string" use="required"/>
    <xsd:attribute name="version" type="sim:versionType" use="required"/>
  </xsd:attributeGroup>
  <xsd:simpleType name="versionType">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="[0-9]+\.[0-9]+"/>
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:complexType name="simulationType">
    <xsd:sequence>
      <!-- components involved in the simulation from the view point of the client -->
      <xsd:element name="componentInstances">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="componentInstance" type="sim:componentInstanceType"
              maxOccurs="unbounded"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="composition" type="sim:compositionType"/>
      <!-- simulation results to be collected from components -->
```

```

<xsd:element name="results">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="status" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="componentName" type="xsd:string"/>
            <xsd:element name="variableName" type="xsd:string"/>
            <xsd:element name="variableType" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
<xsd:attribute name="sessionName" type="xsd:string" use="required"/>
</xsd:complexType>
<xsd:complexType name="componentInstanceType">
  <xsd:sequence>
    <xsd:element name="setupVariable" maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="name" type="xsd:string"/>
          <xsd:element name="type" type="xsd:string"/>
          <xsd:element name="value" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="operationRule" type="sim:ruleConfiguration" minOccurs="0"
      maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attributeGroup ref="sim:componentAttributes"/>
</xsd:complexType>
<xsd:complexType name="compositionType">
  <xsd:sequence>
    <xsd:element name="entityFlows">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="entityFlow" type="sim:entityFlowType"
            maxOccurs="unbounded"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="variableFlows">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="variableFlow" type="sim:attributeFlowType"
            maxOccurs="unbounded"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>

```



```

<xsd:complexType name="entityFlowType">
  <xsd:sequence>
    <xsd:element name="entityLink" maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="source" type="xsd:string"/>
          <xsd:element name="targets">
            <xsd:complexType>
              <xsd:sequence>
                <xsd:element name="target" maxOccurs="unbounded"/>
              </xsd:sequence>
            </xsd:complexType>
          </xsd:element>
          <xsd:element name="routingRule" type="sim:ruleConfiguration"
            maxOccurs="unbounded"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
  <xsd:attribute name="entityType" type="xsd:string" use="required"/>
</xsd:complexType>
<xsd:complexType name="attributeFlowType">
  <xsd:sequence>
    <xsd:element name="provider" type="xsd:string"/>
    <xsd:element name="subscribers">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="subscriber" maxOccurs="unbounded"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string" use="required"/>
</xsd:complexType>
<xsd:complexType name="ruleConfiguration">
  <xsd:sequence>
    <xsd:element name="configVar" type="sim:configurationVariableType"
      maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string" use="required"/>
  <xsd:attribute name="type" type="xsd:string" use="required"/>
  <xsd:attribute name="uri" type="xsd:string"/>
  <xsd:attribute name="applyToEntity" type="xsd:string" use="required"/>
</xsd:complexType>
<xsd:complexType name="configurationVariableType">
  <xsd:choice>
    <xsd:sequence>
      <xsd:element name="type" type="xsd:string"/>
      <xsd:element name="value" type="xsd:string"/>
    </xsd:sequence>
    <xsd:sequence>
      <xsd:element name="objectName" type="xsd:string"/>
      <xsd:element name="attributeName" type="xsd:string"/>
    </xsd:sequence>
  </xsd:choice>
  <xsd:attribute name="name" type="xsd:string" use="required"/>
</xsd:complexType>
</xsd:schema>

```

Appendix D

Entity Type File

```
<?xml version="1.0" encoding="utf-8"?> <!-- edited with XMLSpy
v2005 U (http://www.xmlspy.com) by Xinjun Chen (NTU) -->
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:ent="http://www.fcbdsg.org/entity"
targetNamespace="http://www.fcbdsg.org/entity"
elementFormDefault="unqualified"
attributeFormDefault="unqualified">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      </xsd:documentation>
    </xsd:annotation>
    <!-- The root element -->
    <xsd:element name="entity" type="ent:entityType"/>
    <!-- Definition of Global element "description" -->
    <xsd:element name="description" type="xsd:string"/>
    <!-- Definition of Global simpleType "versionType" -->
    <xsd:simpleType name="versionType">
      <xsd:restriction base="xsd:string">
        <xsd:pattern value="[0-9]+\.[0-9]+\.[0-9]+"/>
      </xsd:restriction>
    </xsd:simpleType>
    <xsd:complexType name="entityType">
      <xsd:sequence>
        <!-- Description of the entity type -->
        <xsd:element name="description" type="xsd:string"/>
        <xsd:element name="attribute" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="name" type="xsd:string"/>
              <xsd:element name="type" type="xsd:string"/>
              <xsd:element name="defaultValue" type="xsd:string"
minOccurs="0"/>
              <!-- Description of a single attribute -->
              <xsd:element name="description" type="xsd:string"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:sequence>
</xsd:schema>
```

```
<xsd:attribute name="type" type="xsd:string" use="required"/>
<xsd:attribute name="provider" type="xsd:string" use="required"/>
<xsd:attribute name="version" type="ent:versionType" use="required"/>
</xsd:complexType>
<xsd:element name="attribute"/>
</xsd:schema>
```

Appendix E

Papers Published/Submitted from this Research

1. W.B. Zong, Y. Wang, W.T. Cai, and S.J. Turner. Grid Services and Service Discovery for HLA-Based Distributed Simulation. In Distributed Simulation and Real-Time Applications (DS-RT 2004), pages 116 - 124, October 2004.
2. X. J. Chen, W. T. Cai, S. J. Turner, and Y. Wang. SOAr-DSGrid: Service-Oriented Architecture for Distributed Simulation on the Grid. In 20th Workshop on Principles of Advanced and Distributed Simulation (PADS 2006), pages 65 - 73, Singapore, May 2006.
3. Y. Wang, S.J. Turner, W.T. Cai, and X.J.Chen. Time Management in a Service-Oriented Architecture for Distributed Simulation on the Grid. In Summer Computer Simulation Conference (SCSC 2007), pages 392 - 399, July 2007.
4. X.J. Chen, W.T. Cai, S.J. Turner, and Y. Wang. Shared Variable Management in SOAr-DSGrid. In Second Asia International Conference on Modelling and Simulation (AMS 2008), pages 154 - 161, April 2008.
5. Y.Wang and S.J. Turner. Rule Execution in a Service-Oriented Architecture for Distributed Simulation on the Grid. In Asia Simulation Conference 2008/7th Inter-

national Conference on System Simulation and Scientific Computing (ICSC 2008),
October 2008. (Submitted).