Framework for component-based distributed simulation on the grid : specification and coordination

Chen, Xinjun

2009

Chen, X. (2009). Framework for component-based distributed simulation on the grid : specification and coordination. Master' s thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/18701>

<https://doi.org/10.32657/10356/18701>

NANYANG TECHNOLOGICAL UNIVERSITY

A Framework for Component-based Distributed Simulation on the Grid: Specification and Coordination

Chen Xinjun

School of Computer Engineering

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

2009

ABSTRACT

The emergence of Grid computing is a milestone in high performance computing. The Grid computing promotes flexible, secure, coordinated, and massive resource sharing among networks. Component technology boosts code reusability. The service-oriented architecture is an ideal paradigm for next generation computing. The loose coupling among services in the service-oriented architecture relieves service consumers from detailed knowledge of implementation, implementation language, and execution platform of the services to be consumed. It is innovative and advisable to take advantages of both the Grid computing and component technology and apply these technologies to distributed simulation in a service-oriented architecture. A framework for component-based distributed simulation on the Grid comes into being with the above considerations.

The framework facilitates the development and execution of distributed simulation in different views, i.e., a component-based view for development and a service-oriented view for execution. At component-development level, the framework standardizes the simulation components and simulation applications in order for interoperability between components and reusability of the existing components. A simulation component conforms to a standard component interface, and a simulation application conforms to a pre-defined simulation description schema. At service-execution level, each component is executed as a Web Service in Globus Toolkit version 4. A pre-deployed organizer service is employed to initialize a simulation application, to dispatch the execution-time information to relevant components, and to collect simulation result. The framework supports shared variable management in distributed simulation by introducing history list and future list.

This thesis details the framework pertaining to the specifications, the design of the architecture and its components, automatic service generation, runtime implementation of

the framework, and different approaches for shared variables management. The profiling on an automatically generated service is carried out. An experiment on the shared variables management is conducted to compare the performances of different approaches in shared variables management. In addition, a case study of wafer fabrication is used to illustrate how the above concepts can be applied.

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my gratitude to those individuals who have given me a lot of support for this project.

Foremost, I would like to express my heartfelt thanks to my supervisor, Dr. Cai Wentong, who has imparted his most invaluable knowledge and experience on numerous fields to me. Have it not for his patience in teaching, guiding, and supporting me throughout the course of this project, this research would not have been possible.

Special thanks go to Dr. Stephen J. Turner for his constructive probing questions and suggestions, invaluable support, and encouragement during the course of the project. His inspiring questions moot some ideas of the project to me.

I would also like to extend a special note of gratitude to Mr. Wang Yong for his collaboration in the project.

Finally yet importantly, I would like to express my appreciations to Ms. Irene Goh, Senior Technician of Parallel and Distributed Computing Centre (PDCC), for her assistance with the GT4 configurations and other technical support.

TABLE OF CONTENTS

ABSTRACT	I
ACKNOWLEDGEMENTS	III
TABLE OF CONTENTS	IV
LIST OF FIGURES	VII
LIST OF TABLES	IX
LISTINGS	X
CHAPTER 1	1
INTRODUCTION	1
1.1 BACKGROUND	1
1.1.1 <i>Distributed Simulation and HLA</i>	1
1.1.2 <i>Service Oriented Architecture (SOA) and Grid Technologies</i>	3
1.2 OBJECTIVES	6
1.3 ORGANIZATION OF THE THESIS	8
CHAPTER 2	10
LITERATURE REVIEW	10
2.1 COMPONENT TECHNOLOGIES	10
2.2 COMPONENT-BASED SIMULATION	14
2.3 SERVICE WORKFLOW MODELS	18
2.4 SHARED VARIABLES MANAGEMENT	22
2.5 SUMMARY	23
CHAPTER 3	25
FRAMEWORK OVERVIEW	25
3.1 CONCEPTS AND DEFINITIONS	26
3.2 FRAMEWORK OVERVIEW	30
3.2.1 <i>Component Provider</i>	31
3.2.2 <i>Client</i>	33
3.2.3 <i>Organizer</i>	35
3.3 SUMMARY	36
CHAPTER 4	38
SCHEMAS AND INTERFACES	38

MASTER THESIS

TABLE OF CONTENTS

4.1 PROVIDER INTERFACES	38
4.1.1 <i>Entity Type File</i>	39
4.1.2 <i>Component Interface Description File</i>	40
4.1.3 <i>Composite Component Implementation File</i>	42
4.1.4 <i>Rule Type File</i>	45
4.2 CLIENT INTERFACES	48
4.2.1 <i>Simulation Description File</i>	48
4.3 CASE STUDY: WAFER FABRICATION SIMULATION MODEL	49
4.4 SUMMARY	54
CHAPTER 5	55
COMPONENT SERVICES DEVELOPMENT	55
5.1 DEVELOPING COMPONENT AS A SERVICE	55
5.1.1 <i>Component Resource</i>	56
5.1.2 <i>Component Resource Home</i>	65
5.1.3 <i>Component Service Implementation</i>	65
5.2 FACILITIES FOR COMPONENT DEVELOPMENT	70
5.2.1 <i>General Code Generation</i>	70
5.2.2 <i>Service Generator</i>	71
5.3 SHARED VARIABLE MANAGEMENT	75
5.3.1 <i>Data Structures and Web Service Interfaces</i>	75
5.3.2 <i>Basic Mechanisms</i>	78
5.4 EXPERIMENTS	82
5.4.1 <i>Component Service Generation and Profiling</i>	82
5.4.2 <i>Shared Variable Experiments</i>	86
5.5 SUMMARY	90
CHAPTER 6	92
SIMULATION LIFECYCLE	92
6.1 SIMULATION INITIALIZATION	92
6.2 SIMULATION EXECUTION	96
6.3 SIMULATION TERMINATION	98
6.4 SUMMARY	101
CHAPTER 7	102
CONCLUSIONS AND FUTURE WORK	102
7.1 CONCLUSIONS	102
7.2 OUTLOOK OF FUTURE WORK	105
APPENDICES	106
APPENDIX A : SCHEMAS	106
A.1 <i>Entity Type Schema</i>	106

MASTER THESIS

TABLE OF CONTENTS

A.2	<i>Component Interface Schema</i>	106
A.3	<i>Composite Component Implementation Schema</i>	108
A.4	<i>Simulation Description Schema</i>	110
A.5	<i>ServiceGen Schema</i>	112
A.6	<i>Intermediate XML File Schema</i>	115
APPENDIX B	: SAMPLE FILES	116
B.1	<i>OperationA (Owner's Operation)</i>	116
B.2	<i>CompA Component Interface</i>	119
B.3	<i>Shared Variable Simulation Description File</i>	120
B.4	<i>CompA ServiceGen</i>	121
B.5	<i>Sample Intermediate XML File</i>	121
APPENDIX C	: TEMPLATES	122
C.1	<i>Resource Template</i>	122
C.2	<i>ResourceHome Template</i>	123
C.3	<i>Service Implementation Template</i>	124
C.4	<i>WSDD Template</i>	125
C.5	<i>JNDI Template</i>	126
C.6	<i>WSDL Template</i>	127
APPENDIX D	: WAFER FABRICATION FILES	129
D.1	<i>EtchStrip Component Interface</i>	129
D.2	<i>CMP Component Interface</i>	132
D.3	<i>Wafer Fabrication Simulation File</i>	134
APPENDIX E	: GRAPHIC USER INTERFACES	140
E.1	<i>Service Generator GUI</i>	140
E.2	<i>Index Service GUI</i>	141
E.3	<i>Organizer Service GUI</i>	142
APPENDIX F	: GRID METHODS DESCRIPTION	143
F.1	<i>PrimitiveComponent Grid Methods Description</i>	143
F.2	<i>Organizer Grid Methods Description</i>	144
PUBLICATIONS		145
BIBLIOGRAPHY		146

LIST OF FIGURES

Figure 1.1	Create a Resource in WSRF	5
Figure 1.2	Use a Resource in WSRF.....	6
Figure 2.1	CCA port connection between components.....	13
Figure 3.1	Definitions and relationships	28
Figure 3.2	SOAr-DSGrid Framework Overview	30
Figure 3.3	Layered Structure of SOAr-DSGrid	31
Figure 3.4	Component development procedure	32
Figure 3.5	Simulation development procedure	33
Figure 3.6	Client's procedure in simulation	34
Figure 3.7	Simulation initialization.....	36
Figure 4.1	Entity type schema.....	39
Figure 4.2	Component interface schema.....	40
Figure 4.3	Composite component implementation file schema.....	42
Figure 4.4	Operation rule in composite component implementation file schema.....	43
Figure 4.5	Entity flow in composition	44
Figure 4.6	Variable flow in composition.....	45
Figure 4.7	Simulation description file schema.....	49
Figure 4.8	Wafer fabrication processes	50
Figure 5.1	Logical structure of PrimitiveComponentResource	56
Figure 5.2	Class diagram of PrimitiveComponentResource	57
Figure 5.3	Movements of entities.....	60
Figure 5.4	Primitive component manager flowchart.....	63
Figure 5.5	Entity and operation selection flowchart	63
Figure 5.6	Class diagram of PrimitiveComponent.....	69
Figure 5.7	General code generation model	71
Figure 5.8	Class diagram of ServiceGeneratorEngine	72
Figure 5.9	Component service generation process.....	73
Figure 5.10	Data structures and interface for shared variables.....	77
Figure 5.11	Sequence diagram for shared variable update.....	79
Figure 5.12	Round trip of a grid invocation	84
Figure 5.13	Experiment results (varying reader unit processing time).....	88

MASTER THESIS

LIST OF FIGURES

Figure 5.14	Experiment results (varying request pattern)	89
Figure 6.1	Class diagrams of the organizer service and organizer resource	93
Figure 6.2	Initialization process of a simulation	94
Figure 6.3	Flowchart of simulation start in component resource.....	96
Figure 6.4	Sequence diagram for the premature simulation termination	100
Figure 6.5	Sequence diagram of the normal simulation termination	100

LIST OF TABLES

Table 2.1	Comparison of COM, CORBA, EJB, and CCA	14
Table 2.2	Comparison of SimBean, DEVS/COM, SimKit	16
Table 2.3	Relationship between WSFL, BPEL4WS, and WSCI/BPML	21
Table 2.4	Comparison of WSFL, GSFL, and BPEL4WS	21
Table 5.1	createResource getResource profiling results	85
Table 5.2	Simulation factors	86
Table 5.3	Simulation configuration (varying reader unit processing time)	87
Table 5.4	Simulation configuration (varying request pattern)	89

LISTINGS

Listing 4.1	Sample routing rule type GreaterThanOrEqualToBranch.....	46
Listing 4.2	Sample routing rule configuration.....	47
Listing 4.3	Rule file LessThanBranch.rule.....	51
Listing 4.4	CMP machine routing rule snapshot	52
Listing 4.5	Code fragment of wafer manufacturing simulation description file	53
Listing 5.1	Common service implementation.....	66
Listing 5.2	Interfaces for the organizer.....	67
Listing 5.3	Interfaces for peer component services	67
Listing 5.4	Fossil-collection of the history list	82
Listing 5.5	Fossil-collection of the future list.....	82

CHAPTER 1

INTRODUCTION

1.1 Background

A simulation model is “*an approximation, representation, or idealization of selected aspects of the structure, behavior, operation, or other characteristics of a real-world process, concept, or system*” [46]. Simulation is defined as the reproduction of a system with the use of simulation models (hereinafter, “model” refers to “simulation model” if not explicitly specified). Through the reproduction, real-world problems can be identified without incurring the cost of running the physical system, and different solutions can be proposed and tested on the simulation system without affecting the physical system. Simulation is often desirable especially when it is of high-cost or danger to carry out the testing on the physical system.

1.1.1 Distributed Simulation and HLA

Distributed simulation, together with parallel simulation, is the joint between simulation community and high performance computing community. Distributed simulation executes a model on loosely-coupled networked computers. These computers may be geographically distributed across buildings, universities, or countries. Any low-cost personal computers, possibly with heterogeneous platforms, can be involved in distributed simulation if they are connected to the network.

Three elements are crucial in designing a distributed architecture. These include object/component interface language, object/component management, and naming service [10]. With respect to distributed simulation, one additional element is the time management.

Every object/component should bear an *interface* that is known to all the participating objects/components and the specific framework. A common standard interface format is an agreement that every participating object/component is obliged to observe when providing its interface. Participating objects/components interact with each other with reference to the corresponding interfaces.

Object/Component manager creates/removes an object/component in/from the framework. If objects/components can interact or communicate directly, object/component manager also needs to pass the remote references of desired objects/components to the requesting ones either at simulation initialization phase or at run time upon requests. Object/Component manager may also take the responsibility of coordinating the objects/components.

Naming service, also known as directory service or index service, maintains a registry of published objects/components. It enables registration, deregistration, update, and query of objects/components.

Time management, also called *synchronization algorithms*, is not required in general distributed architecture. However, it is a primary element in distributed simulation, because a distributed simulation cannot execute properly without a properly synchronized simulation clock. Synchronization algorithms [33, 34] have been well researched from late 1970's in parallel and distributed simulation community.

High Level Architecture (HLA) [41] is a widely used distributed architecture in defense simulation. It was developed by US Defense Modeling and Simulation Office (DMSO) in 1995 and was standardized as IEEE 1516, as a replacement of its predeces-

sors (Distributed Interactive Simulation (DIS) [42] standards and Aggregate Level Simulation Protocol (ALSP) [71, 73]). The concepts of interoperability and model reuse introduced in SIMNET [57] project have been addressed in HLA.

HLA consists of three major components: HLA Rules [44], HLA Federate Interface Specification [43], and Object Model Template (OMT) [45]. HLA only specifies the services and interfaces to Runtime Infrastructure (RTI). DMSO provides a reference implementation of RTI software [22]. There are also other versions of RTI available, such as Pitch's pRTI [58] and Georgia Tech's FDK [28].

HLA provides a complete and robust standard for the development of small to large distributed simulation. However, there are some drawbacks. Firstly, the complexity of the HLA RTI programming interfaces leads to steep learning curve. RTI specifies more than 130 functions, of which developers need to learn and use a quite significant subset. Secondly, the RTI only provides a single level of federation. This results in the lack of information hiding among federates in the same federation [12]. Thirdly, it is not easy to compose federates/federations into a composite one. The composition of some federates into a composite federate requires substantial work such as merging federate ambassadors, mapping instance variables etc. Fourthly, the performance of the RTI is not optimized due to the substantial processing when updates of instance attributes happen [51]. Finally, a joined federate cannot directly communicate with another joined federate in the federation. All communication between cooperating federates, either instance attribute updates or interactions, must be carried out via the RTI.

1.1.2 Service Oriented Architecture (SOA) and Grid Technologies

Service Oriented Architecture (SOA) is originally driven by the globally changing business. Nowadays, business models change more frequently than IT systems. Most organizations expect to transform their enterprises more responsively to the change of the business models. Thus, they need to reorganize information resources into substantially inde-

pendent and reusable services. These independent services are more adaptable to the changing business environment. The independence and reusability features make SOA expand to other areas like research, education, and etc.

The Web Services [74] is a runtime implementation of these independent and reusable services. Web Services are implemented based on open standards. A Web Service is described by Web Services Definition Language (WSDL) [75], an XML-based language. A WSDL file contains the information about the service endpoint URL, the communication protocols, the interface (i.e., the operations that the Web Service can perform), and the messages formats. The communication between a Web Service and its client is through Simple Object Access Protocol (SOAP) [63] format messages. The transport protocol is transparent to the Web Service implementation, and it can be HTTP, JMS, or even SMTP, etc. The open standards allow the client to use Web Services irrespective of the client's technology and application environments. Therefore, Web Services make applications and systems from different vendors easily integrated in the enterprise environments.

The concept of Grid computing was proposed by Ian Foster et al. in the mid 1990s [30, 31]. Surveys showed that worldwide computing resources were idle by a large proportion. The Grid was thus proposed to utilize these idle computing resources. The Grid promotes a *“flexible, secure, coordinated resource sharing among dynamic collections of individuals, institutions, and resources”* (referred to as virtual organizations or VOs) [31]. It denotes a distributed computing infrastructure for advanced science and engineering.

Grid technologies are gaining more and more attention within the IT industry as well as scientific research. The Open Grid Services Architecture (OGSA) [32], developed by the Open Grid Forum¹ (OGF) [37], defines an open architecture for grid-based applications. It defines Grid services and a set of interfaces for them. Web Services are the underlying technology for Grid services. Grid services bring in added features such as life-

¹ Open Grid Forum (OGF) was formed in 2006 as the merger of the Global Grid Forum and the Enterprise Grid Alliance.

cycle management, service data, notification, portType extension, statefulness, and transience of services. All the Grid service features are technically specified in the Open Grid Services Infrastructure (OGSI) [65], which defines mechanism for creating, managing, and exchanging information between Grid services. Globus Toolkit version 3 (GT3) [38] provides a complete implementation of OGSI.

In Jan 2004, a standard, Web Services Resource Framework (WSRF) [77], was published as a more polished and stable standard that substitutes OGSI. It partitions functionalities in the monolithic OGSI into five specifications [21]: WS-ResourceLifetime, WS-ResourceProperties, WS-RenewableReferences, WS-ServiceGroup, and WS-BaseFault. Notification in OGSI is specified by WS-Notification specification series. OGSI promotes factory/instance pattern for grid services. It introduces service state by adding resource state into traditional stateless Web Services. WSRF preserves the state, but also preserves the advantages of scalability of stateless Web Services. It decouples the Web Service and the resource state. Therefore, the Web Service and resource are clearly separately in WSRF, and the composition of these two is called WS-Resource. Each WS-Resource is addressed and accessed by a WS-Addressing endpoint reference, which is the combination of the service URL and a resource identifier. A WSRF-compliant Web Service consists of a service implementation, a resource, and a resource home. The relationships between the three terms are illustrated in Figure 1.1 and Figure 1.2.

Figure 1.1 Create a Resource in WSRF

Figure 1.2 Use a Resource in WSRF

When a service client invokes a Web Service, the service client ensures the availability of the resource instance by invoking the `createResource()` operation before invoking any other operations. Upon receiving the `createResource` request, the service implementation will ask resource home to create a new instance of resource. An endpoint reference is returned to the service client. The endpoint reference contains both the address of the service and the identifier of the resource. The endpoint reference information is carried in the context data for all consecutive invocations. When a Web Service invocation reaches the service implementation, the service implementation first tries to find a resource instance from the resource home based on the invocation context. The resource home finds the correct resource instance based on the resource identifier. The service implementation operates on the resource instance returned.

The release of WSRF results in the convergence of Grid services and Web services. The GT4 [38] is a comprehensive implementation of WSRF, including GT4-Java, GT4-C, and pyGridWare. Other implementation of WSRF includes WSRF::Lite [79] from University of Manchester and WSRF.NET [78] from University of Virginia.

1.2 Objectives

The emergence of Grid computing is a milestone in high performance computing. It

makes sense to take advantages of the Grid and apply the Grid technologies to distributed simulation so that distributed simulation can be carried out on the heterogeneous platforms on the Grid. In this project, the proposed framework will lie on top of the Grid.

The development and implementation of a simulation model is always very complex and time-consuming. In order to fully take advantage of distributed simulation and make it widespread in industrial and commercial areas, we need to tackle two major issues.

Firstly, the complexity of model construction must be simplified. Modeling is of paramount importance in distributed simulation. The high-degree of complexity of model development may hinder the adoption of HLA in commercial and industrial areas. The complexity of modeling primarily results from the complexity of the physical system to be modeled, which we can hardly change. Another important factor lies in the framework of the simulation software. Properly designed simulation framework could make modelers' life easier.

Secondly, the cost of distributed simulation should be minimized. Industry users are cost-oriented and emphasize Return of Investment (ROI). Simulation will be carried out only if it makes economic sense, i.e., the benefit from simulation outcores the cost required in terms of money and time to develop the model. It is apparently advisable to reuse previously well-developed and tested whole or part of simulation models.

Based on the above analysis, we propose a Service-Oriented Architecture for Distribution Simulation on the Grid (SOAr-DSGrid). In this framework, a complex model can be divided into smaller simple simulation components. A simulation application is constructed by orchestrating the existing simulation components. Within this framework, a component provider develops and publishes simulation components. He/She also provides a component interface for each component published for ease of reuse. The underlying implementation of the component is in a service-oriented way. A client can build a simulation model by reusing the existing simulation components and assembling them in

a simulation description file. The details about the roles of the component provider, the client, and other modules in the framework will be covered in Chapter 3.

Componentizing the simulation model is an effective way to promote reuse. This framework makes the reuse more flexible since both component providers and clients are able to work at an individual component level instead of the entire simulation. The framework also standardizes the interfaces for component and simulation development, and the development procedures for component providers and clients.

We propose this component-based approach also because it enables collaborative model development. Component developers from different sites and different organizations can develop their simulation components independently. The component-based approach allows them to work in a collaborative way in which the components developed by each component developer can be used collaboratively to form simulation models.

The framework proposed is novel and original to the best of the author's knowledge. The author's contributions include the specification of components, the specification of simulations, the specification of service interfaces, the design and implementation of the runtime, automatic service generation, and the evaluation of different approaches for share variables management. The framework covers the basic aspects of simulation such as modeling mechanism, simulation execution strategy, time management, composition, etc. Some other aspects of simulation, such as semantic validation, composability, etc are also important, but are outside the scope of this thesis.

1.3 Organization of the Thesis

The rest of the thesis is organized as follows. Chapter 2 reviews the component technologies, and the research that has been carried out in component-based simulation, the workflow execution models, and shared variable management. Chapter 3 defines relevant terms and provides an overview of the architecture of the proposed framework. Chapter 4

elaborates the framework interfaces that are used by component providers and clients. Most of the interfaces have the corresponding XML schemas. Chapter 5 details how a component can be provided as a service and how the shared variable support is developed with some experiments. Chapter 6 illustrates the simulation lifecycle and details how a simulation execution is orchestrated. Chapter 7 concludes the thesis, summarizes the features of the framework, and outlines the future work.

CHAPTER 2

LITERATURE REVIEW

This chapter takes surveys on component technologies, component-based simulation, service workflow, and shared variable management. The proposed framework realizes component-based simulation by employing service orchestration. The proposed framework also supports shared variable management.

2.1 Component Technologies

Component technologies have been actively explored by both simulation community and software engineering community. We believe the component technology is more mature in software engineering community than in simulation community. In Component-Based Software Engineering (CBSE), there are several famous definitions on component [40, 48, 64]. CCA Forum, a High Performance Computing association, also has a formal definition of software component [13]. A software component is a software object, a standard unit of applications. From these definitions, we determine that a simulation component should bear the following features:

- Well-designed and clearly-defined interface,
- Conformance to a specific component model,
- Independent deployment, and
- Ease of composition.

The above component features are also design considerations of our framework as our framework is developed to support component-based modeling and simulation. These features are also reflected in the framework overview in Chapter 3 and in the framework interfaces design in Chapter 4.

CBSE has been matured to a certain level. The three most popular component models are Microsoft's Component Object Model (COM) [18], Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) [20], and Sun Microsystems' Enterprise JavaBeans (EJB) [23]. All the three models enforce their own specific binary structures. All of them borrow some idea from Open Software Foundation's² (OSF) Distributed Computing Environment (DCE) Remote Procedure Call (RPC) [62] protocol. A client component call is first sent to a local proxy or object request broker (ORB), where the function name and parameters are marshaled to a format suitable for network transmission and transferred to a server component. The remote proxy, ORB, or skeleton unmarshals the function call and forwards the restored function name and parameters to the server component for computation. The computation result is first sent to the server-side proxy, which marshals the result and sends it to the calling component. The proxy of calling component then receives and unmarshals the result, and forwards it to the calling component procedure. The interfaces are exposed as stubs.

COM is Microsoft's technology for communication between software components within a process or amongst processes in a local machine. DCOM extends the support to a network. COM+ is an extension to COM and DCOM, and it has been integrated with the Microsoft Transaction Server (MTS). Currently COM technologies are only prevailing in Windows platforms, though Microsoft has released a version of COM/DCOM for MacOS. COM is claimed as language-neutral among Microsoft languages such as Visual C++, Visual Basic, etc.

² Open Software Foundation is now named as Open Group. The website is [http:// www.opengroup.org](http://www.opengroup.org).

OMG provides an abstract interface definition language (IDL) [19], which enables application objects developed in different languages to be interoperable. A CORBA compliant system must at least adhere to CORBA Core and provide a mapping from IDL to a programming language. CORBA is an architecture, i.e., specification, but not implementation. OMG provides an official implementation of CORBA, namely, CORBA Component Model (CCM) [16]. CORBA is a quite complex specification, and requires expertise from developers.

In Sun's EJB, both business logic and business data are componentized as session beans and entity beans respectively. The components developed using EJB are interoperable in any platform with Java Virtual Machine. However, the composition of beans is not simple, not to mention hierarchical composition of beans.

US Department of Energy (DoE) led the Common Component Architecture (CCA) initially for large-scale scientific parallel computation. It is a component architecture that defines standards necessary for the interoperation of components developed in different frameworks. CCA specification [4, 15] defines the interfaces which are implemented by every framework complying with the architecture. A CCA port is a functional unit that defines interactions between components. CCA promotes a *provides/uses* design pattern. As shown in Figure 2.1, every component has some "provides" ports (implementation of the class or functions of a port) and "uses" ports (function calls to "provides" ports of another component). A provides port can be analog to a Grid service, and a uses port is analog to a client of a Grid service. The communication between components is carried out through ports after port instances of the same type have been connected.

Figure 2.1 CCA port connection between components

This provides/uses design pattern makes the component connection easy to be realized dynamically as the connection of components can be carried out at run-time instead of component development time. The dynamic component connection is termed as composition in space [39]. However, CCA is oriented at scientific computing, so the components are generally stateless. There is no specification about component state and component state information sharing between components. This drawback is inherited in its reference implementations such as Ccaffeine [14], SCIRun [84], and XCAT [49, 80].

The eXtreme Component Architecture Toolkit version 3 (XCAT3), developed by Indiana University eXtreme lab, is a framework which allows CCA components to be compatible with OGSI specifications. In XCAT3, every provides port is implemented as a Grid service, and every component is implemented as a set of Grid services since a component may have multiple provides ports [49]. This enables XCAT3 components to run on the Grid. Another project in the eXtreme lab, Application Factory Web Service (AFWS) [3, 36], provides a means to compose the existing components into an application. However, neither XCAT3 nor AFWS supports WSRF. They are both based on OGSI and Globus Toolkit 2 and 3.

Another drawback is that neither XCAT nor AFWS provides module for time management, which is quite crucial in distributed simulation. Moreover, the ports exportation

mechanism in AFWS is so simple that it does not handle the complex composition of ports. Table 2.1 summarizes the above-mentioned technologies in terms of the languages they support, whether they support dynamic composition at runtime, whether they support hierarchical composition (i.e. whether a composite component can be constituted by other composite components), and whether the technology have time management support.

Table 2.1 Comparison of COM, CORBA, EJB, and CCA

	COM	CORBA	EJB	CCA
Languages	Microsoft Languages	C, C++, Java, Smalltalk, COBOL, Ada, Lisp, PL/1, Python	Java	Fortran77, Fortran90, C, C++, Java, Python
Dynamic composition	No	No	No	Yes
Hierarchical composition	No	No	No	No
Time management support for simulation	No	No	No	No

To conclude, COM, CORBA, and EJB, are oriented towards commercial software development. CCA is oriented towards general scientific computing. CORBA and CCA are language-neutral. Most of the existing technologies do not support dynamic or hierarchical composition of components. None of them provides a mechanism for time management and thus they are not suitable for component-based distributed simulation.

2.2 Component-based Simulation

Contrasting to the general component technologies, component-based simulation is still in its infancy, which may be because CBSE is supported by commercial organizations. In addition, there are more specific issues that need to be tackled in the simulation area, such as time management, hierarchical modeling, model validation, and syntactic and semantic composition.

In [83], Zeigler introduced theoretic Discrete Event System (DEVS) formalism to provide a means to model discrete event systems in a hierarchical modular way. With this DEVS formalism, we can model a large system by decomposing it into component mod-

els and specifying the coupling between them. The two kinds of models in DEVS formalism are atomic model and coupled model. An atomic model specifies a set of states and state transition function. A coupled model specifies its components and its coupling scheme. The atomic model is analogical to atomic/primitive components, and coupled model is analogical to composite components. This modular hierarchical system modeling approach can also be used in simulation modeling.

In SimBean [59, 60], Praehofer et al. proposed a simulation framework based on JavaBeans. The framework includes five layers in top-down order: application simulation tools, application specific simulation components, basic simulation components, simulation kernel, and Java/JavaBeans extensions for simulation. The whole architecture is based on the event-driven simulation worldview. A delegate is used to listen to events. The delegate invokes the target method of a target object when processing events. A delegate is the counterpart of the message manager in our framework. The simulation kernel provides infrastructure for simulation execution, including event scheduling and event handling. Some basic simulation components like queues and processors are generally reusable. Application-specific components can be developed to model some concrete real-world entities like machines and vehicles by assembling basic simulation components. An application-specific simulation system can be developed by the orchestration of application specific components. SimBean supports modular hierarchical modeling by connecting input and output interfaces of individual components. SimBean emphasizes the discrete event model, though it also supports continuous and combined simulation mechanisms.

In [17], Cho and Kim presented a DEVS framework for discrete event modeling and simulation using COM component technology. They suggested combining the advantages of DEVS and COM technology. However, their paper does not describe how COM technology is incorporated in the DEVS framework. Nevertheless, their work is the first at-

tempt to build DEVS framework based on COM architecture. Before that, Wang and Ho have used Java component technology to develop DEVS-JavaBean [69]. Zeigler et al. has also tested distributed simulation on DEVS/CORBA execution environment [82].

SimKit [8, 9] is the implementation of Listener Event Graph Objects (LEGOs) [11] component-based framework from US Naval Postgraduate School. A basic event graph is used to represent scheduling relationships between events. The authors create the concept of event graph objects. These event graph objects are treated as atomic components representing a process. LEGO also employs a listener pattern to link event graph objects by an edge that represents listener relationships. We can find that this is essentially similar to the workflow that we will discuss in the next section. However, as can be seen from [8, 9], the nodes in the event graph are events that will happen at certain time with certain conditions. There is no intuitive mapping from the node to a simulation entity or resource. Thus, event graph objects are more suitable to model a process, but not simulation entities and resources.

Table 2.2 compares the SimBean, DEVS/COM, and SimKit by the following criteria: 1) whether it is an active project; 2) whether it supports simulations of continuous systems and; 3) whether it supports distributed simulation. Only SimKit is an active project. All of them can be used for discrete event simulation. SimBean can also be used for continuous system simulation and combined system simulation. One fatal disadvantage is that none of them is implemented to support distributed simulation.

Table 2.2 Comparison of SimBean, DEVS/COM, SimKit

	SimBean	DEVS/COM	SimKit
Is active project	No	No	Yes
Continuous system support	Yes	No	No
Distributed simulation support	No	No	No

As XML is extensible, platform independent, and expressive to human reader, it is also introduced into component-based simulation. Simulation Reference Markup Lan-

guage (SRML) [61] can be used to describe general-purpose simulations and it is essentially an XML schema. It adds behavior to XML through scripts. “The goal of SRML is to enable simulations or other contents to be served, received, and processed in a standard fashion using Internet technologies and the World Wide Web” [61]. Though it is not dedicated for component-based simulations, the SRML allows the inclusion of external items, which can be separately compiled component-based objects.

With the increase of simulation complexity, component-based simulation faces the issues of how to effectively reuse existing simulation parts and how to compose existing simulation parts by commonly agreed upon message interfaces. Simulation Interoperability Standards Organization (SISO) proposed the Base Object Model (BOM) [5] to address these issues. A BOM template is an XML file that contains piece-parts of simulations including model identification, conceptual model, model mapping, HLA object model, notes, and lexicon definitions. The model identification contains meta-data about the component. The conceptual model describes the types of actions and events in the component. The model mapping maps the conceptual entities and events to their HLA object model representations. The HLA object model contains the information about objects, attributes, interactions, and parameters, which are included in normal FOM/SOM. The structure of notes is the same as that in the HLA OMT templates. The lexicon definitions include conceptual model definitions and object model definitions. With the BOM, a federation developer can select the existing BOMs from a repository and compose them into a simulation.

Moradi et al [24-27] explored the SRML and BOM and used both in component-based simulation. In [25], a model developer formulates the intent of a simulation in SRML, and a tool (XVCL) will identify the BOMs that are needed in the simulation from the existing repository. The selected BOMs are further selected to satisfy semantic composition requirement using ontology information. The finally selected BOMs form a BOM assembly,

which is used to generate FOM and SOM. In [26], the authors introduce semantic BOM attachment (SBA) to the BOM by including ontology for entities, events, and interactions. They use different rules at different layers of composibility stack (the syntactic layer, the static semantic layer, and the dynamic semantic layer). In [27], the authors extend the work of SBA and use agent-based environment for composition. The JACK agent framework is used in the implementation. The manager agent coordinates among the agents. It invokes a parser agent to parse a SRML file. The result (SRML Java bean) is passed to the discovery agent to find necessary BOMs from the repository and to further filter the selected BOMs. The combination agent is used to combine the selected BOMs into different compounds. The different compounds are sent to the composition agent for assessment and scoring. The comparison agent sorts the score list according to the logic defined by the simulation modeler. The final result is a BOM assembly. Introducing semantics into component-based simulation makes the approach viable even if different components use different concepts and terminologies.

2.3 Service Workflow Models

A service workflow is a process that contains a set of services and protocols by which these services interact with each other. A workflow specifies a series of tasks to be executed in a specific time and in a logical sequence. We focus on discussion of workflow languages for Web services and Grid services.

Web Services Flow Language (WSFL)

Business transactions often involve multiple invocations of Web Services. Web service framework alone does not realize these complex interactions or compositions of Web services. WSFL [7, 76] is proposed by IBM as a solution to this problem.

WSFL uses a flow model to represent the flow of a process. The elements specified in a flow model include the service providers involved in the process, the activities, the con-

trol flow, and the data flow. Activities are operations in the flow, and are specified by input, output, and internal or external implementation. Control flow specifies execution order of the activities. A control link in control flow presents a link between a source activity and a target activity with a transition condition. Data flow specifies how the output messages of service are used by others. A data link in data flow specifies a source activity, a target activity, and a message mapping. Besides the above four elements, the flow model also supports flow model input data by flow source element, and flow model output data by flow sink element.

Our proposed framework borrows the concept of data flow in WSFL. Yet, we separate the data flow into entity flow and variable flow. The data link in WSFL links two activities, while entity link in our framework links two component instances. The variable flow in our framework represents the variable sharing between component instances.

The global model in WSFL denotes a concrete composition request. It describes a complex transaction. The global model contains a set of service providers and plug links. A service provider declares an instance of service provider with associated service provider types. A plug link specifies the interaction between two service instances by a source-target operation pair. The counterpart of global model in our framework is the simulation description file.

WSFL 1.0 was released in May 2001. However, there is no much effort on WSFL since this initial release. The reason may fall on a critical drawback of WSFL: two service providers cannot communicate directly at run time. All data transfer between them must be intermediated by a WSFL engine. This centralized data flow may be a bottleneck for applications with bulk data transfer between individual services. Even with improved network connection, the centralized data flow in WSFL is still considered a source of inefficiency of data transfer.

Grid Services Flow Language (GSFL)

GSFL [50] was jointly developed by US Argonne National Laboratory and Indiana University. It was the first attempt to export the WSFL idea to Grid service workflow. GSFL borrows many ideas from WSFL, and tailors them to the needs of Grid services in the OGSI framework.

GSFL also uses concepts of service providers, activities, control links, data links which are similar to those in WSFL. The service locator now can specify either a Grid service instance or Grid service factory with a Grid Service Handler (GSH). Notification model addresses communication between Grid services by notification links. The runtime data flow in GSFL models is decentralized. The notification model and Grid service notification feature are the prerequisite of peer-to-peer data transfer. However, the development of GSFL is heavily dependent on Globus Toolkit (GT). As admitted by the GSFL inventors, many changes took place in GT and made GSFL not feasible.

Business Process Execution Language for Web Services (BPEL4WS)

BPEL4WS [6, 70] is an initiative of BEA Systems, IBM, Microsoft, SAP AG, and Seibel Systems. BPEL4WS is the combination of earlier works in IBM WSFL and Microsoft XLang. It can be used to implement executable business processes as well as describe non-executable abstract processes. BPEL Web services are not created in a factory/instance pattern. An instance is created implicitly at run time with some information within data messages if the instance is not available at that time.

However, BPEL4WS also has some drawbacks. One drawback is that BPEL4WS only supports input-only or input-output (request-response) operations [72], but not output-only (notifications) or output-input (solicit-response) operations. This means BPEL4WS has no callback or notification mechanisms. This is apparently a vital disadvantage when dealing with distributed simulations.

Another drawback is that BPEL4WS has some overlapped primitives, as BPEL4WS is

the combination of WSFL and XLANG. Aalst [1] gives examples that different activities/primitives are used to realize the same workflow patterns. This kind of redundancy increases the complexity of BPEL4WS engines.

Sun, BEA, SAP, and Intalio also introduced another candidate for services composition: Web Service Choreography Interface (WSCI). Each WSCI document specifies interactions or message exchanges of one Web service. WSCI only specifies the public interactions or choreographies of Web services, which is alike the public interface in object-oriented design and analysis, but no internal implementations. The private internal implementation is usually developed by another meta-language Business Process Management Language (BPML). BPML has incorporated WSCI protocol for description of public interaction or choreographies. Aalst et al. [2] provide in depth comparison between BPEL4WS, XLANG, WSFL, and WSCI using both workflow patterns and communication patterns. The following Table 2.3 summarizes the relationship between the above-mentioned specifications.

Table 2.3 Relationship between WSFL, BPEL4WS, and WSCI/BPML

	Public Interaction	Private Implementation
WSFL	Global Model	Flow Model
BPEL4WS	Abstract Processes	Executable Processes
WSCI/BPML	WSCI	BPML

As shown in Table 2.4, most of the workflow languages support hierarchical composition, but none of them is suitable for simulation due to the lack of time management. Only GSFL has notification mechanism to support peer-to-peer data transfer.

Table 2.4 Comparison of WSFL, GSFL, and BPEL4WS

	WSFL	GSFL	BPEL4WS
Hierarchical composition support	Yes	No	Yes
Notification mechanism support	No	Yes	No
Time management support	No	No	No

2.4 Shared Variables Management

The shared variable issue has existed and been researched in the simulation area for more than a decade. As early as 1993, Mehl and Hammes [56] proposed two general approaches to implementing shared variables in a conservative synchronization algorithm, namely the request-reply and the cached-copy. In the request-reply, the owner maintains a history list for each shared variable. A history list contains the records which may be requested by the readers. The reader sends a request message to the owner for the value of the shared variable at simulation time t , and the owner will not send back the value until it is sure that the value of the shared variable will not be changed before t . The reader is time-constrained by the owner, but not vice versa. The reader always issues a remote request when the variable is needed. In the cached-copy, the reader maintains a cached copy of the shared variable associated with the time-guarantee information, which indicates the validity period of the cached-copy. The reader sends a request message to the owner only if the requested time goes beyond the validity period.

Most parallel discrete event simulation (PDES) protocols either forbid the use of shared variables or avoid the shared variable problem by certain partitioning techniques. In [66], Turner et al. proposed to partition simulation entities into separate conflict sets. In manufacturing simulation, if multiple machines share an operator, all these machines and the associated shared operators will be partitioned into the same conflict set. Each conflict set is then assigned to a logical process (LP). Hence, there are no shared variables between conflict sets (i.e., LPs).

In [35], Fujimoto proposed to allow zero-lookahead in the HLA time management specification to allow instantaneous access to attributes of a federate. Thus, the retrieval of shared variables can be implemented using zero-lookahead request and reply TSO messages. However, both the owner and the reader federates must be time-constrained and must regulate the federation with a lookahead of zero. This is detrimental to the per-

formance of the simulation. In order to improve the performance, Low et al. [53] revisited the shared variable problems, implemented, and conducted experiments on four Receive Order (RO) approaches, namely PullRO, PullROTG, PushRO, and PushROTG. Since RO messages are not used for computation of lookahead and time advancement constraint, the owner federate can regulate the federation with a non-zero lookahead. In the PullRO approach, the owner maintains a history list and the reader maintains a cache. The reader sends pull request to the owner if the cache copy is no longer valid. In response, the owner will return the value with the time at which the value is updated. In PullROTG, the response from the owner also includes the time guarantee information to indicate the validity period of the cache. In the PushRO approach, the reader maintains a future list which contains the records that may be used by the reader in the future. The owner will push the value to the reader whenever the shared variable is updated. The reader will issue pull request, if the required value cannot be found from the future list. In PushROTG, the owner pushes the value with time guarantee information to the reader.

If the state sharing exists in a distributed simulation, the shared variables are required. State sharing is common in many distributed simulations. Avoiding the shared variable will make the simulation model more complex, more difficult to understand and maintain. In particular, shared variable support is required in our proposed framework as to be explained in Chapter 3. We will propose different approaches based on PullROTG and PushROTG in this thesis. Different from PullROTG, in which only the requested record is returned to the owner, the new approach will greedily return all the available records to the reader as much as possible. Different approaches are compared in Chapter 5.

2.5 Summary

The aforementioned component architectures aim at distributed software or scientific applications, and the aforementioned component-based simulation projects address some

simulation issues. The workflow languages aim at business processes management.

COM, CORBA, and EJB are not customized for simulation. CCA and XCAT have no support for sophisticated composition. AFWS port connection is rather simple and does not provide complex hierarchical component or application composition. Neither XCAT nor AFWS support WSRF and were implemented before Grid services and Web Services converged. SimBean, DEVS/COM, and SimKit are suitable for discrete event simulation on standalone computer. They neither provide nor reside on a middleware that supports distributed simulation. GT3/GT4 itself has no support for Grid/Web Service composition. WSFL has centralized data flow and is outdated. GSFL died on the vine due to the constant change of the Globus Toolkit. BPEL4WS is complex and is mainly developed for business process management.

The author is trying to find a technology/project that has the following characteristics:

- enables distributed simulations;
- enables the execution in the Grid environment;
- supports complex model construction and simulation application construction via composition of simple finer simulation components;
- separates concerns of modelers and simulation application developers; and
- addresses state sharing issues in distributed simulation.

However, to the best of the author's knowledge, none of the existing systems satisfies all the criteria. Therefore, in the following chapters, we present a framework for distributed simulation in the Grid environment that aims to satisfy the above criteria.

CHAPTER 3

FRAMEWORK OVERVIEW

In this thesis, we propose a Service Oriented Architecture for executing Distributed Simulation on the Grid: SOAr-DSGrid. In this architecture, two different views are presented on the simulation development and simulation execution. The component-based view is used in the simulation development. To expose the operations implemented in a simulation component and to facilitate the composition of a simulation, schemas are defined for the component developers to publish component interfaces and for the simulation application developers to describe the constituent components and their interactions in a simulation.

The service-oriented view is presented during the simulation execution. Each simulation component is implemented as a GT4 service by extending some base component service modules. To hide the details of service and simulation management implementation, the base component service modules provide the underlying simulation management such as event scheduling and time advancement in addition to the basic service implementations. Component services communicate with each other through pre-defined service interfaces and they are configured according to the component interactions specified in the simulation description.

The separation of component interface from service interface makes the development of a component as a service easier. Component providers just need to describe the operations of a component using the component interface schema and implement the compo-

ment operations accordingly. They do not need to worry about how a service is implemented, how a service interface is defined, and how services communicate with each other. Simulation application developers can then use the component interfaces and the simulation description schema to compose a simulation application. They do not need to worry about how the component services are located and how they are deployed and invoked.

The use of component schema and the simulation description schema enables collaborative development of simulation applications, and the deployment of simulation components as services takes advantage of the SOA (e.g., loose coupling, heterogeneity, and transport neutrality). Using the component-based view, a simulation application developer can search for the required simulation components, and describe the composition of the components. Under our service-oriented architecture, simulation component services can be deployed to the heterogeneous resources and communicate via various transport layer protocols. They work in a peer-to-peer manner: a component service can be a service provider and a service consumer at the same time.

In this chapter, we give an overview of the framework. The schemas and interfaces are covered in detail in Chapter 4. Chapter 5 will discuss in details how a component can be implemented as a service, and Chapter 6 describes how different services can be orchestrated into a simulation execution.

3.1 Concepts and Definitions

This section defines some terms used in SOAr-DSGrid for the ease of elaboration. We define a **component** as a processing unit that has a specific non-empty set of **operations**. A component possesses a set of **variables**. Variables can be either public or private. **Public variables** are published and exposed to other components, i.e. public variables are sharable. **Private variables** are only accessible from inside the component. The ensemble of

variables represents the *state* of a component.

An *operation* is a functionality of a component, and it performs certain processing on certain type(s) of *entities*. The inputs involved in an operation are variables of the current component, public variables of other components, and/or an entity. All the required information, including the inputs and other required resources, is encapsulated in the implementation of the operation. The output of an operation is either an entity with modified properties or a new entity. The component interface specifies a list of supported operations. The interface of any component in this framework bears a common component interface format, which is elaborated in section 4.1.2.

An *entity* is an object that is processed by components. Each entity has a set of *properties* denoting the state of the entity. During a simulation session, an entity is passed between components to undertake a series of operations, which might modify the properties of that entity. An entity can be generated in two ways. A specific type of components, also known as entity generators, can be developed to generate entities. In addition, some components may generate new entities when performing operations on entities. Another specific type of components, also known as entity collectors, can be developed to collect and destroy entities.

The *route* of an entity is a sequence in which the entity is processed. Each node in the sequence represents a component in which the entity is to be processed. The route information is derived from a simulation description file (to be further discussed in section 4.2.1). Whether the exact route of an entity is determined at simulation design time or run time depends on the simulation scenario. If the route is determined at run time, routing rules will be employed. A *routing rule* is a decision logic that is applied in a component to determine which component is the next destination among all the possible destinations. A routing rule is applied to a component if the next immediate destination of entities can only be determined after processing the entities.

The operation to be performed on an inbound entity can also be determined at run time. An **operation rule** is a decision logic that is used to determine which operation of a component to perform on a certain entity. Apparently, operation rules are only applicable if the component has more than one operation.

Routing rules and operation rules can be provided either by a component provider at component development time or by a client at the simulation application development time. A rule contains **decision logic** and a **configuration**. A configuration assigns values to variables used in the rule description file, and links the variables used in the rule description file to the variables of local or remote components, or to the attributes of entities. A **local rule engine** is used to parse rules and replace the configurable variables with concrete values or references at run time. More explanation on rules is available in section 4.1.4.

Figure 3.1 Definitions and relationships

Finally, we introduce some terms to describe the relationships between components. There are two categories of communications between components, i.e., variable sharing

and entity transfer. The two components are called *linked component* of each other if they have any of the above two relationships. If a component transfers entities to another component, the former is termed as the *preceding component* or *source component*, also known as *regulating component* in time management, and the latter is termed as the *following component* or *target component*, also known as *constrained component* in time management. If a component subscribes the shared variable of another component, the former is termed as the *provider component*, and the latter is termed as the *consumer component* or *subscriber component*. More details on component relationships can be found in section 4.1.3 when entity links and variable links are discussed.

Figure 3.1 illustrates the terms involved in the framework with examples. *Component CB* has three *variables*: a *private variable* $v1$, a *public variable* $v2$ (which is subscribed by component *CA*), and a *private variable* $v3$, (which is capable of accepting initialization value, i.e., *setup variable*, from the simulation description file). *Entity $e1$* has *attribute $a1$* , and is processed by component *CA*.

Component *CA* produces new entity $e2$ based on entity $e1$. A component can also simply modify some attributes of an entity instead of producing a new one. Component *CA* will send entity $e2$ to component *CB* or *CC*. *Routing rule $r1$* is employed by component *CA* to determine the next component of entity $e2$. The public variable $v2$ of component *CB* may be used when component *CA* executes rule $r1$.

With respect to component relationships, Component *CA* has two linked components, i.e., *CB* and *CC*, both are following components. Component *CA* has one provider component, i.e. *CB*. The preceding component(s) of *CA* is not shown in the figure. Component *CB* has one preceding component, i.e., *CA*, and one subscriber component, i.e., *CA*. Component *CC* has one preceding component, i.e., *CA*. Both *CB* and *CC* has only one linked component, i.e., *CA*.

3.2 Framework Overview

As shown in Figure 3.2, there are five roles in the architecture: the component provider (i.e., simulation component developer), the client (i.e., the simulation application developer), the repository, the organizer, and the index service. The component provider develops simulation components. The client builds simulation applications.

Figure 3.2 SOAr-DSGrid Framework Overview

The repository stores the relevant files including component interface files, entity type files, and rule type files. Component interface files and entity type files which contain descriptive information about components and entities, are used during the development of a simulation application.

In this service-oriented architecture, the organizer, the index service, and every simulation component are implemented as GT4 services. The organizer acts as an agent to the client. It receives simulation description files submitted by the client, executes the simula-

tion application, and returns the simulation results to the client. A centralized index service provides directory information on components, component interface files, entity type files, and rule type files. This directory information is required for component development and application development.

Figure 3.3 illustrates the layered structure of the entire framework. SOAr-DSGrid is a layer residing between the users and GT4. The parts contained in the dotted box in the middle represent the schemas and GUI provided to the component providers and clients. The solid parts are used by the component providers, the stripped parts are used by the clients, and the meshed part (GUI) is used by both the component provider and the client. The details about the interfaces and schemas are covered in Sections 4.1 and 4.2. Other parts in the middle layer form the SOAr-DSGrid runtime. The component runtime includes base modules for generating simulation component services and tools for configuring and deploying component services. Chapter 5 elaborates the component runtime and addresses the issue of how the runtime links to the interfaces.

Figure 3.3 Layered Structure of SOAr-DSGrid

3.2.1 Component Provider

Each simulation component provides a specific functionality or a set of related functionalities. The component provider's main task is to implement these functionalities, and make them consumable in SOAr-DSGrid.

Figure 3.4 Component development procedure

Figure 3.4 demonstrates the procedures of component development by component providers. The component provider identifies all the variables and operations of the component, determines the entity types to be processed, and conceives some rules which may be required by the component. The component provider may create new entity types conforming to entity type schema (refer to section 4.1.1) if the entities involved in component operations are of new entity types. If new rule types are required, the component provider also creates necessary new rule types (refer to section 4.1.4).

The component interface file is used to generate a WSRF Web Service for the corresponding component. SOAr-DSGrid provides Graphics User Interface (GUI) for the component provider to generate, build, and deploy the component service. Subsequently, the component provider deposits and registers the entity type files, the rule type files, and

the component interface files respectively. Finally, the URI of the deployed component service is registered with the index service.

3.2.2 Client

Figure 3.5 Simulation development procedure

Before the client develops a simulation, he/she queries the index service to locate the component interface files, the entity type files, and the rule type files separately and in succession as shown in Figure 3.5. In turn, the index service returns the location information and the client uses that information to retrieve the respective files from the repository. With the help of these files, the client constructs a simulation description file, which conforms to the simulation description file schema. If the simulation involves new rules types, the client also creates the required new rule type files, deposits them to the repository, and registers the locations of the rule type files.

Figure 3.6 Client's procedure in simulation

Figure 3.6 shows the procedure of a client in running a simulation. When a client runs a simulation, he/she first queries the URI of the organizer service if he/she does not know its location. The client then invokes the organizer service to create an organizer resource instance. The organizer returns the resource instance information to the client, and all the subsequent client requests to the organizer will contain the resource instance information in the request context. After the organizer resource instance is ready, the client submits the simulation description file to the organizer and then instructs the organizer to start the simulation. In consequence, the organizer will initialize the simulation and trigger the execution of all the component instances. The details on how the organizer works are covered in Chapter 6. During the simulation, the client can query the index service about the URI of each component instance. The component instance information can be used to

query the execution status of that instance. At the end of the simulation, the organizer will return results to the client. As a result, the client will remove the organizer instance.

3.2.3 Organizer

The organizer acts as the agent for the clients to execute simulation applications. It is a GT4 web service responsible for processing simulation description files. It initializes a simulation session, but does not intervene in the execution of simulation components.

The simulation initialization process is shown in Figure 3.7. Clients locate the existing organizer service by querying the index service. They create a resource instance (according to WSRF) for the organizer and submit a simulation request. The organizer parses the simulation description file and creates a simulation session accordingly.

The organizer queries location information of participating component services and instantiates a resource for each participating component service. Subsequently, the organizer initializes the participating component services by passing them the relevant information such as setup variables, linked components, and shared remote variables. Each participating component service subscribes to its shared remote variables of the respective owners. If a rule type file is not available locally, it will be downloaded from the repository using the directory information obtained from the index service. After finishing all the setup procedures, the organizer activates all the participating component services to start simulation. Chapter 6 will provide more details regarding how organizer is implemented.

Figure 3.7 Simulation initialization

3.3 Summary

SOAr-DSGrid involves five roles: component provider, client, organizer, index service, and repository as shown in Figure 3.2, and is on top of GT4 container. Schemas are provided for both the component provider and the client, and a GUI is also developed. The component provider uses the GUI of the service generator and the index service. The cli-

ent uses the GUI of the index service and the organizer. The component runtime, the organizer, and the index service form the SOAr-DSGrid runtime. The simulation design is in a component-based approach for component reuse, and the simulation execution is in a service-oriented approach for loose coupling and flexibility.

CHAPTER 4

SCHEMAS AND INTERFACES

The schemas and interfaces are designed in the way that the proposed framework adopts the process-interaction simulation world-view. The process-interaction simulation world-view presents a system from the active entities' point of view. The active entities travel through the system.

4.1 Provider Interfaces

In order to build a distributed framework capable of accepting new components and orchestrating these components into an application, it is imperative to standardize the component interfaces and the scheme of component interactions. In this framework, a standardized component interface is presented in section 4.1.2. Primitive components and composite components bear the same interface format. This enables the client to treat primitive components and composite components in the same way. Most of the interactions between components are based on entities, which travel between components and are processed by components. Thus, it is also necessary to standardize the format of an entity type.

In this framework, the formats of component interface files, composite component implementation files, simulation description files, and entity type files are all defined using XML Schema [81].

4.1.1 Entity Type File

An entity type is simply a data structure in implementation. Figure 4.1 shows the structure of an entity type. Appendix A.1 contains the detailed XML schema of an entity type.

Figure 4.1 Entity type schema

An entity type has its *type*, *provider*, *version*, *description* about the entity, and a set of *properties*. Each entity type file in the repository defines one entity type. Attribute *type* is the name of the entity type. Attribute *provider* may be used in access control for entity type maintenance. Attribute *version* is reserved for entity type maintenance. Each *property* element has a *name*, *type*, and some simple *description* about the property element. The dotted box represents an optional element. If the default value of a property element is not “0” or “null”, it can be specified by *defaultValue*. In addition to the description for the entity, there is also a description for each property element. The rich description enables efficient selection of entity types. Sufficient description is required in both entity type files and component interface files for reuse by others. An entity type is published by a component provider. It is usually created and published together with the first primitive simulation component which can process that type of entities.

4.1.2 Component Interface Description File

A component interface file is created by a component provider when developing a component, and is used by a client to build a simulation model or used by a component provider to develop a composite component. A component interface file serves as documentation providing a clear picture of the component, and thus promotes component reuse. Figure 4.2 is the pictorial view of the component interface schema (refer to Appendix A.2 for the complete component interface XML schema).

Figure 4.2 Component interface schema

The concept of component interface in SOAr-DSGrid is different from that in general software development. Application Programming Interface (API) in general software de-

velopment provides a way to access attributes of objects/components and to perform interactions between objects/components. However, as elaborated at the beginning of Chapter 3, SOAr-DSGrid presents a component-based view for simulation development and a service-oriented view for simulation execution. In the component-based view, a component interface is more about the data structure of a component, what kinds of variables can be shared between components, and what kinds of entities can be processed or generated by the component. The simulation application developer is not required to know the details about the component implementation and the low-level service API that only presents in the service-oriented view. The separation of component interface and service interface enables the simulation application developer to concentrate on component orchestration and composition instead of runtime aspects of the component, such as the service implementation, the location, and the deployment of the components.

Referring to Figure 4.2, a component has some profile information. Attribute *type* specifies the name of the component type. Attribute *version* and Attribute *provider* are similar to those of an entity type schema in Figure 4.1. Attribute *isComposite* denotes whether or not this type of component is composite. Attribute *category* indicates whether the component is an entity generator, entity processor, or entity collector.

A component has a set of *variables* and *operations*. Each *variable* has its *name*, *type*, and *description*. The component provider specifies the default value of a variable if it is not “0” or “null” using *defaultValue*. Element *mode* specifies whether this variable is accessible to other components (*localShared*), is a pure local variable (*local*), or is a local copy of a remote shared variable (*remoteShared*). The component interface also specifies the *name* and the *description* of every operation. For each operation, the entity types involved are also specified. Element *inputEntity* specifies all the entity *types*, entity instances of which can be processed by this operation. Element *outputEntity* specifies all the entity *types*, entity instances of which can be generated by this operation.

4.1.3 Composite Component Implementation File

A composite component is much more complicated than a primitive component. In addition to a component interface file, every composite component has a composite component implementation file, which describes the constituent components of the composite component and how these constituent components are composed. The runtime support for composite component is one area of the future work.

A composite component implementation file is created by a component provider when developing a composite component. It is used by a composite component manager at run time to decompose the composite component.

Figure 4.3 depicts the structure of the XML schema used by composite component implementation files. Appendix A.3 gives the complete XML schema for composite component implementation files.

Figure 4.3 Composite component implementation file schema

A composite component implementation file consists of three major parts. First part is the profile of the composite component: the *name*, *type*, *version*, and *provider* of the composite component. The second part is instance information of its constituent components. The last part is the composition information including the entity flows and variable flows.

Every component instance possesses a *name*, *componentType*, and *version*. Component instance information also includes the initialization values of its setup variables. Notice that *setupVariables* are a subset of *variables* of the component specified in component interface in Figure 4.2. Every setup variable is initialized by a tuple of *name*, *type*, and *value*.

Figure 4.4 Operation rule in composite component implementation file schema

A component instance also specifies the operation rules it will use at run time if this component is a multi-functional component. Figure 4.4 shows the structure of a rule configuration. Attribute *type* specifies the rule type and attribute *name* identifies a rule configuration. Optional attribute *uri* specifies the URI of the rule type. This URI can also be obtained at run time by querying the index service with the rule type as input parameter. For any component instance, there is at most one operation rule for each type of entity.

Attribute *applyToEntity* specifies the entity type to which this rule applies. The element *configVars* stores configurable variables. A *configVars* could be assigned with a *value* of certain type, instance of certain component, or specified as a reference to variable *variableName* of *objectName*. The *objectName* could be either the name of a component instance or “*currentEntity*” which refers to the current entity under operation. The rule configuration is designed in this way with the consideration that the configurable variables may be different for different types of entity instances. Listing 4.2 in section 4.1.4 gives an example of rule configuration.

Figure 4.5 depicts the entity flows in composition. An **entity flow** is the flow of a particular type of entity in a simulation model. Each entity type, specified by *entityType*, has one *entityFlow*. Each *entityFlow* specifies one *entryComponent* and a set of *entityLinks*. Whenever an entity is transferred to a composite component, it is actually passed to the *entryComponent* of the corresponding flow.

Each *entityLink* specifies transition of an entity from a component *source* to component *target(s)*. The entity can be sent to different *target* components based on the routing rule. If there is only one *target* in *targets*, no *routingRule* shall be specified. Otherwise, *routingRule* must be provided. The *routingRule* has the same structure as *operationRule* depicted in Figure 4.4 (complex type *impl:ruleConfiguration*).

Figure 4.5 Entity flow in composition

A *variable flow* specifies the publish-subscribe relationship of one public variable of the component instance. Each *variableFlow* specifies one *provider* who publishes the *variableName* and a set of *subscribers* (refer to section 3.1).

Figure 4.6 Variable flow in composition

Notice that information recorded in the entity flows and variable flows is internal to the composite component. The values of *entryComponent*, *source*, *target*, *providerName*, and *subscriberName* are all component instance names under element *componentInstances* in the composite component implementation file. One exception is the element *target*. The value of element *target* could also be “external” if the *entityLink* specifies a link from the composite component to the outside world.

4.1.4 Rule Type File

Two kinds of rules are employed in the framework, namely operation rules and routing rules. The purpose of rules is to realize flexible dynamic execution inside a component instance and dynamic interaction between simulation component instances. Inside a component instance, operation rules are employed to determine which operation to perform. A routing rule determines the flow of a certain type of entities when there are multiple routes available. Both rules have the same rule configuration format (refer to Figure 4.4) and rule type format. A rule is decoupled from a component. This is why component interface files contain no information on rules.

A rule type file is created by a component provider when he/she develops a component which requires a new rule type, or created by the client when he/she constructs a model and needs a new rule type. A rule type file is used together with a rule configuration by a component at run time to determine which operation to perform, or to which component instance the current entity should be sent.

An important part of rule configuration is the variable configuration which maps the variables used in the rule to certain values or references:

- Threshold values
- References to variables representing the current state of the component to which the rule is applied
- References to public variables representing the current states of other components in the composition
- References to attributes representing the current state of the entity to be processed or routed

The input to a rule will be generated based on the rule variable configuration during runtime. The output of a rule is the name of the next component for a routing rule, and the operation name for an operation rule.

Generally, a rule type file contains some conditional branch statements with a set of variables. Listing 4.1 presents a sample routing rule type `GreaterThanOrEqualToBranch`. The format of an operation rule is the same as that of a routing rule.

Listing 4.1 Sample routing rule type `GreaterThanOrEqualToBranch`

```
if $variable1 >= $THRESHOLD
    return $aComponentName;
else
    return $anotherComponentName;
```

As shown in Listing 4.1, a rule may use several configurable variables. A variable used in a rule may be mapped to a threshold value (e.g. `$THRESHOLD`), public variable of another component (e.g. `$variable1`), or a component name (e.g. `$aComponentName`).

The mapping is specified in the rule configuration. Listing 4.2 shows a sample configuration for the rule type listed in Listing 4.1. The rule configuration in Listing 4.2 conforms to the schema (complex type *impl:ruleConfiguration*) depicted in Figure 4.4. The combination of the rule logic presented in a rule type file and the rule configuration presented in a simulation description file or a composite component implementation file uniquely determines the operation of a rule within a simulation session.

At run time, the rule engine of current component parses this rule logic together with the configuration and checks if the *utilization* of component *InstanceA* is greater than or equal to 90%. If yes, the current component will send current entity of *ATypeEntity* to component *InstanceB*; otherwise, it will send it to component *InstanceA*.

Listing 4.2 Sample routing rule configuration

```
<routingRule name="AComponentExitRule"
type="GreaterThanOrEqualToBranch" applyToEntity="ATypeEntity">
  <configVars name="variable1">
    <objectName>InstanceA</objectName>
    <attributeName>utilization</attributeName>
  </configVars>
  <configVars name="THRESHOLD">
    <type>float</type>
    <value>0.9</value>
  </configVars>
  <configVars name="aComponentName">
    <type>ComponentB</type>
    <value>InstanceB</value>
  </configVars>
  <configVars name="anotherComponentName">
    <type>ComponentA</type>
    <value>InstanceA</value>
  </configVars>
</routingRule>
```

Rule type file is also subject to reuse. Potentially, rule types can be reused in the following ways:

- 1) The creation of a rule type is independent of the components which use the rule. Therefore, different component instances of different component types could use the rules of the same rule type.
- 2) In the design of rule configuration, introduction of attribute *applyToEntity*

makes a rule type reusable for different types of entities.

4.2 Client Interfaces

The framework provides the client with some interfaces for simulation modeling. The most important interface for the client is the XML schema for simulation description files. The rest of this section discusses this schema. Other interfaces for the client are the XML schemas for rule type files and rule configuration. They are the same as the rule type file and the configuration format for the component provider (refer to section 4.1.4 for details).

4.2.1 Simulation Description File

A simulation description file is created by a client as the result of simulation model design. It is used by the organizer at run time to execute the simulation. Figure 4.7 depicts the structure of a simulation description file. Appendix A.4 gives the complete XML schema for the simulation description file.

The format of a simulation description file is quite similar to that of a composite component implementation file as both files describe the orchestration or composition of components. This design also implies that a composite component manager and the organizer have something in common.

There are three differences between the formats of a simulation description file and a composite component implementation file. The most salient difference is the addition of return results in the simulation description file. Some accounting variables of certain components may be specified as return results. This is equivalent to sharing some public variables of simulation components to the organizer. Every returned variable is specified with a tuple *componentName*, *variableName*, and *variableType*.

Two other minor differences are the simulation profile and entity flows.

- The profile of simulation file only specifies a simulation session name with at-

tribute *sessionName*.

- In the entity flow, no *entryComponent* is specified as all entities are within a simulation session. This is different from a composite component which receives entities from other components.

Figure 4.7 Simulation description file schema

4.3 Case Study: Wafer Fabrication Simulation Model

A case study on wafer fabrication simulation modeling is conducted using the framework

interface. In order to build a simulation model, the analysis of the physical model is a prerequisite. The flow diagram shown in Figure 4.8 is synthesized from [47] to the best of our understanding.

Figure 4.8 Wafer fabrication processes

The first machine produces wafer from raw semiconductor material. The operations this machine can perform include “MakeWafer” and “Epitaxy”. The machine can only perform one operation at a time. Therefore, a certain amount of time (machine setup time) is required for a multi-functional machine to switch from one operation to another. The wafer produced by the first machine will travel to the subsequent machines in the production line. In this model, we assume the processes “Oxidation”, “Photolithography”, “Etching”, “Stripping”, “Diffusion”, “Implantation”, “Deposition”, “Metalization”, and “CMP” are looped three times to generate source, gate, and drain of a transistor respectively. After that, the wafer is sent to the back-end line for testing and assembling. In this model, we use the testing component alone to represent the whole back-end operations for simplicity.

A machine is modeled as a component and is represented by a box in Figure 4.8. Dotted boxes represent the operations supported by the machine. In general, each component has one specific functionality. Multi-functional machines are modeled as multi-functional components, e.g. the machine which performs etching and stripping operation in Figure 4.8. Operation rules will be used by multi-functional component to determine which operation to perform at run time. The links denote the flow of entities. If a component has multiple outgoing links, e.g. CMP machine, routing rules must be provided for the local

rule engine to decide the next component at run time.

Appendix D.1 is the component interface for EtchStrip machine. The setup variables of a component form a large part of the component interface file. During this modeling, we follow the Modeling Data Standards (MDS) version 0.1 [29]. The relevant setup variables include tool set ID, operator set ID, machine setup time, wafer load and unload times, travel times, process times, the fraction of operator involvement time over the processing time, and etc. The operations are indexed in setup variable names in Appendix D.1. For instance, the name *processTime1* means the processing time of Etching operation, and *processTime2* the processing time of Stripping operation. The Appendix D.1 is edited according to the XML schema for component interface.

Appendix D.2 is the component interface for Chemical Mechanical Planarization (CMP) machine. The CMP operation represents the processing in which an entity (i.e., a wafer) is polished to smooth. A wafer may be sent to the furnace for Oxidation process as the start of another operation loop after the CMP operation. If the loop count of the entity reaches three, the CMP machine will send the wafer to the back-end line, i.e. the tester for testing.

Appendix D.3 shows the wafer fabrication simulation description file. It is edited according to the XML schema for simulation description files. It shows the detailed setup variables of only the first two component instances as the other component instances have similar setup variables.

Listing 4.3 Rule file LessThanBranch.rule

```
if $variable1 < $variable2
    return $exit1;
else
    return $exit2;
```

Listing 4.3 shows the rule type used by CMP component, and Listing 4.4 shows the rule configuration extracted from the simulation description.

In this model, variable *variable1* is the reference to the loop count of the entity cur-

rently being processed. Variable `variable2` is set to integer 3. Variable `exit1` is “oxidationTool” denoting an instance of `OxidationTool` component. Variable `exit2` is “tester” denoting an instance of `Tester` component. The configuration is presented in Listing 4.4.

Listing 4.4 CMP machine routing rule snapshot

```
<routingRule name="CMPRoutingRule" type="LessThanBranch"
applyToEntity="Wafer">
  <configVars name="variable1">
    <objectName>currentEntity</objectName>
    <attributeName>loopCount</attributeName>
  </configVars>
  <configVars name="variable2">
    <type>integer</type>
    <value>3</value>
  </configVars>
  <configVars name="exit1">
    <type>OxidationTool</type>
    <value>oxidationTool</value>
  </configVars>
  <configVars name="exit2">
    <type>Tester</type>
    <value>tester</value>
  </configVars>
</routingRule>
```

The definitions of all the component instances are inside the simulation description file in Appendix D.3. Listing 4.5 shows the code fragment of a simulation description file pertaining to the component instance definitions, entity flows, and variable flows.

As shown in Listing 4.5, *aCMPMachine* can transfer the current wafer entity to *oxidationTool* or *tester*. At run time, *aCMPMachine* will invoke its local rule engine to execute the rule specified in Listing 4.4.

The public variable *isInputQueueFull*, published by *oxidationTool*, is subscribed by *makeMaskTool* and *aCMPMachine* as the latter two component instances require the variable *isInputQueueFull* to decide whether they can transfer wafer entity to *oxidationTool* from their output queues (refer to Figure 4.8 for the flow diagram).

Listing 4.5 Code fragment of wafer manufacturing simulation description file

```

<componentInstances name="makeMaskTool" componentType="MakeMaskTool" version="0.1">
  ...
</componentInstances>
<componentInstances name="oxidationTool" componentType="OxidationTool" version="0.1">
  ...
</componentInstances>
<componentInstances name="aCMPMachine" componentType="CMPMachine" version="0.1">
  ...
</componentInstances>
<componentInstances name="tester" componentType="Tester" version="0.1">
  ...
</componentInstances>

<composition>
  <entityFlows entityType="Wafer">
    ...
    <entityLinks>
      <source>aCMPMachine</source>
      <targets>
        <target>oxidationTool</target>
        <travelTime>0.1</travelTime>
      </targets>
      <targets>
        <target>tester</target>
        <travelTime>0.2</travelTime>
      </targets>
      <routingRule>
        ...
      </routingRule>
    </entityLinks>
    ...
  </entityFlows>

  <variableFlows>
    <provider>
      <providerName>oxidationTool</providerName>
      <variableName>isInputQueueFull</variableName>
    </provider>
    <subscribers>
      <subscriberName>makeMaskTool</subscriberName>
      <variableName>mask_oxiQueueFull</variableName>
    </subscribers>
    <subscribers>
      <subscriberName>aCMPMachine</subscriberName>
      <variableName>cmp_oxiQueueFull</variableName>
    </subscribers>
  </variableFlows>
  ...
</composition>

```

4.4 Summary

This chapter presents all the schemas and interfaces from a component-based view. The schemas for component development and simulation application development enable the component provider and the client to develop components and simulation applications without worrying about the component service implementation and the simulation execution respectively. The subsequent chapters will present the component service implementation and the simulation execution in a service-oriented view. Due to the limitation of time, the runtime for composite component is not implemented. The implementation of the rules will not further discussed as it is presented in another piece of work by Wang Yong [67].

CHAPTER 5

COMPONENT SERVICES DEVELOPMENT

In the previous chapters, we presented the component-based view on component development and simulation application development. In order to develop a component, the component developer needs to develop a component interface that conforms to the component interface schema and provides the implementation of the supported operations. However, the component interface and the operations are not directly executable. Certain mechanisms are required to make a component executable. In SOAr-DSGrid, we construct each component as a WSRF Web Service to make it invocable and executable over the network. SOAr-DSGrid provides facilities for component developers to generate, build, and deploy a WSRF Web Service for a component. The discussion on shared variables management further illustrates the development of component services.

5.1 Developing Component as a Service

At the component design phase, each component bears a component interface conforming to a schema. At runtime, each component in SOAr-DSGrid is presented as a WSRF Web Service. We term the runtime representation of a component as a *component service* to differentiate it from the design phase representation of a component. As explained in section 1.1.2, each WSRF Web Service has three basic parts, i.e., a resource, a resource home, and a service implementation. To facilitate the component service development, we define

three base classes, i.e., `PrimitiveComponentResource`, `PrimitiveComponentResource-Home`, and `PrimitiveComponent`, for the three basic parts of a component service in SOAr-DSGrid. All component services in SOAr-DSGrid are extended from these three base classes. The following subsections will detail the three basic parts of a component service.

5.1.1 Component Resource

Component resource is a key part of a component service. The component resource enables the statefulness of a component service. We abstract the common aspects in all component resources, and term them as *component-general elements*. Meanwhile, we also allow different component resources to have different variables and operations, and term them as *component-specific elements*. The set of the variables forms the state of a component resource. Different instances of a component resource hold different states. SOAr-DSGrid defines `PrimitiveComponentResource` as the base component resource. The `PrimitiveComponentResource` is designed in such a way that enables support for both component-general and component-specific elements. Figure 5.1 depicts the logical structure of the `PrimitiveComponentResource`, and Figure 5.2 is the corresponding class diagram.

Figure 5.1 Logical structure of `PrimitiveComponentResource`

Figure 5.2 Class diagram of PrimitiveComponentResource

As shown in Figure 5.1, the `PrimitiveComponentResource` comprises both component general elements (denoted by the solid-filled boxes) and component-specific elements (denoted by the mesh-filled boxes). The external entity queue, the internal entity queue, the output entity queue, the output queue manager, the primitive component manager, the internal event queue, the time manager, the rule engine, the shared variable notifier, and the notification processor are the common elements for various component resources. The differences between different kinds of component resources lie in the variables and the operations. The variables describe the properties of the component resource, and they are considered as the state of the component resource. The operations describe the functionalities of the component service. The operations are used to process different types of input entities and generate output entities accordingly. These entities are also considered as component-specific elements. However, they do not belong to any component service. Instead, they travel between component services.

State

The state is the full set of the variables inside a component resource. As shown in the `PrimitiveComponentResource` class in Figure 5.2, some variables are common to all component resources. These include *time* (the simulation time of the component resource), *regulatingComponentTimes* (the simulation times of all the regulating components), *status* (the execution status of the current component resource), *componentEPRs* (end-point references of all the linked component resources), the topology information such as *precedingComponents*, *followingComponents*, *providerComponents*, and *consumerComponents* (refer to section 3.1 and 4.1.3 for the concepts of these terms). The *precedingComponents* refer to a list of names of the preceding component resources. In *followingComponents*, the travel time is specified for each following component resource name and for each entity type. In *providerComponents*, for each provider component resource, a list of shared variables subscribed by current component resource are specified. In *con-*

sumerComponents, for each consumer component resource, a list of its subscribed variables, which are shared variables from current component resource, is specified. The *sharedVariables* maps a local variable name to a *SharedVariable* object. The *subscribedVariables* maps a local variable to a *SubscribedVariable* object. (Details on shared variables are discussed in section 5.3). The *currentEntity* and *currentOperation* are references to the current entity under processing and the current operation respectively.

The state is classified as component-specific because the variables are proprietary to a certain type of component resource. As illustrated in section 4.1.2, each component may have different variables defined in the component interface file. Some of these variables may be local variables, one subset of which is setup variables; some may be public variables that are shared with other components; others may be subscribed variables that are the local copies of the shared variables of other provider components. The definitions of these variables will be generated into a specific component resource file when a component service is generated. The details are illustrated in section 5.2.2. The organizer initializes the setup variables of each component instance based on the values included in the simulation description file as described in section 6.1. All the component-specific variables might be used by the operation of the component.

Operations

The operations contain the simulation logic of a component. SOAr-DSGrid defines an abstract operation class which component developers can extend to define the actual operations of a component. The *execute()* method is used to define the simulation logic of an operation. The *preExecute()/postExecute()* methods are used to define actions to be executed before/after the operation, e.g., reserving/releasing simulation resources, represented by the component-specific state variables. The component resource reference is passed to an operation during its instantiation so that the operation is able to access the state of the component.

Entities and Entity Queues

As illustrated in section 4.1.1, an entity is an object transferred between component resources. Each entity has a timestamp that represents the simulation time when the current component resource receives it. The entities are stored in entity queues in a component resource. There are three entity queues, i.e., the external entity queue, the internal entity queue, and the output entity queue, for temporary storage of entities at different processing stages. The external entity queue stores all the inbound entities that are transferred from other components. The internal entity queue stores the entities that are ready for processing. The output entity queue stores the entities that have been processed by the current component and will be transferred to the next component.

Figure 5.3 Movements of entities

The timestamps of all entities in the external entity queue are greater than the simulation time of the component resource. As shown in Figure 5.3, when the component resource advances its simulation time, the simulation time of the component resource might become equal to the timestamps of some entities in the external entity queue. The time manager will then move these entities to the internal entity queue, i.e., the entities whose timestamps equal to the simulation time of the component resource have arrived at the component and are ready for processing. The entities in the internal entity queue are processed by operations. The primitive component manager may consult the local rule engine to determine which operation should be selected to process an entity. After being processed by an operation, the entity will be moved to the output entity queue. The output

queue manager will invoke the grid method **receiveEntity()** (see Figure 5.6) of the service implementation of the next component, and this will transfer the selected entity in the current component resource's output entity queue to the external entity queue of the next component resource. The same process repeats in the next component resource until the entity reaches an entity collector, which is a special component service that destroys entities.

Time Manager

Time manager³ is a component-general element in a primitive component resource, and it is the implementation of distributed time management in SOAr-DSGrid. The main function of the time manager is to synchronize the simulation and ensure that no time advancement results in causality violation [33]. The time manager class shown in Figure 5.2 implements the algorithm presented in [68]. The time manager is instantiated with the injection of a component resource reference (*resourceRef* in *TimeManager* class in Figure 5.2). The time manager accesses the list of the regulating component times (*regulating-ComponentTimes* in *PrimitiveComponentResource* in Figure 5.2), the list of regulating components, and the list of constrained components through the component resource reference. The simulation time of a regulating component is updated via the **updateTimeAdvancement()** grid method, which forwards the invocation to the *updateTimeAdvancement()* method of the time manager. When the time manager updates its simulation time, it calls its own *informTimeAdvancement()* method, which in turn invokes the **updateTimeAdvancement()** grid method of all its constrained components.

The primitive component manager calls the *nextEventRequest()* method of the time manager to advance the simulation time (the first block in Figure 5.4). During the execution of the *nextEventRequest()* method, the Lower Bound on Time Stamp (LBTS) [68] is calculated using the information of the regulating component times. If the simulation time

³ The algorithm was developed by another M.Eng student, and thus the description here focus on the mechanisms to support time manager rather than the algorithm.

advances to the timestamp of the first event of the internal event queue, an “end of operation” event will be processed. The `postExecute()` method of the corresponding operation will be executed and entities generated by the operation will be also released to the output entity queue with the corresponding timestamp. If the simulation time advances to the timestamp of the first entity in the external entity queue, the entity will be moved from the external entity queue to the internal entity queue. Subsequently, the time manager returns the control to the primitive component manager. If the simulation time does not advance, the execution of the time manager is blocked. It will be waked up when one of the regulating components advances its time via the **`updateTimeAdvancement()`** grid method.

Primitive Component Manager and Entity Processing

The primitive component manager coordinates among other elements, and its responsibility is to select an operation to process entities in the internal entity queue. As shown in Figure 5.4, the primitive component manager marks all the entities in the internal entity queue as unchecked before each invocation to `nextEventRequest()` of the time manager. The primitive component manager can proceed only if the time manager returns the control to it. It then tries to select an entity from the internal entity queue and find its operation. (The search flow is further elaborated in Figure 5.5). If an entity is selected for processing, the primitive component manager removes the entity from the internal entity queue and invokes the `execute()` method of the operation to process the entity. It then schedules an “end of operation” event. If no entity can be processed, i.e., either there is no entity in the internal entity queue, or none of the entities in the internal entity queue can be processed due to unsatisfied resource requirements, the primitive component manager checks whether the simulation is to be terminated. It terminates the simulation by invoking the `terminate()` method of the component if the termination flag is true. Otherwise, it invokes the `nextEventRequest()` method of the time manager again.

Figure 5.4 Primitive component manager flowchart

Figure 5.5 Entity and operation selection flowchart

Figure 5.5 shows the flowchart of entity and operation selection. If there is no un-

checked entity, i.e., either there is no entity in the internal entity queue or all the entities in the internal entity queue have been checked but none of them can be processed due to unsatisfied resource requirement, the selection flow returns without selecting any entity. If there is an unchecked entity, it is selected for inspection. If there is only one operation available for the entity, this operation is selected. If there is more than one operation available, the primitive component manager consults the rule engine to select an operation. It may consult the index service to locate the rule type file and download the rule type file from the repository if the rule type file is not available. After an operation is selected as a candidate, the entity is marked as checked, and the `preExecute()` method of the operation is invoked. If `preExecute()` returns true, the entity and the operation are selected, and the control is returned to the primitive component manager. Otherwise, the next unchecked entity is inspected.

When processing an entity, the operation might access or modify the properties of the entity. It might also access or modify the variables (local variables, local shared variables, or remote shared variables) of the component resource, and/or generates new entities.

Output Queue Manager

The output queue manager selects an entity from the output entity queue, determines the next component resource to which the entity should be transferred. If there are multiple possible next component resources, the output queue manager consults the local rule engine to determine the next component resource. Similar to the primitive component manager in Figure 5.4, the output queue manager may consult the index service and load the rule type file from the repository if the rule type file is not available. The output queue manager probes the component service of the next component to check whether it can send entities to the next component through the grid method `canReceiveEntity()` shown in Figure 5.6. This is to prevent the loss of entities in case the external entity queue of the next component resource is of finite capacity. The output queue manager invokes the grid

method **receiveEntity()** of the component service of the next component to send the entity.

Shared Variable Notifier and Notification Processor

The `SharedVariableNotifier` and the `SharedVariableNotificationProcessor` are two runtime supports for shared variable management. The `SharedVariableNotifier` is responsible for notifying the consumer/subscriber component service of a shared variable record when it becomes available after the current component resource advances its simulation time. The `SharedVariableNotifier` invokes the grid method **receiveVariableUpdate()** of each consumer/subscriber component service to send the new shared variable record. The `SharedVariableNotificationProcessor` is responsible for processing the shared variable records sent by the provider component resource's `SharedVariableNotifier`. The `SharedVariableNotifier`, the `SharedVariableNotificationProcessor`, and other aspects of shared variable management are further discussed in section 5.3.

5.1.2 Component Resource Home

The component resource home is responsible for the creation, discovery, and removal of component resources. The creation of a component resource instance depends on the component resource type, and the component resource instance is casted to the specified component resource type. Each component resource home is dedicated for one specific type of component resource. A sample component resource home is presented in Appendix C.2.

5.1.3 Component Service Implementation

Since there are specifications of entities transfer and variables sharing between components in the component-based view of SOAr-DSGrid (refer to subsection 4.1.2), there must be implementation for such specification in the service-oriented view. The component resource defines a set of variables to denote the state of the component, and a set of

operations supported by the component. However, these variables and operations are not directly accessible by other component services or web service clients. If other component services need to get the shared variable value of the component service, or need to transfer entities to the component service, they need to do it via the web service interfaces exposed by the component service implementation. This subsection will discuss the common web service interfaces exposed by the component service implementation.

Listing 5.1 Common service implementation

```
public class CompA extends PrimitiveComponent {
    private CompAResource getResource() throws RemoteException
    {
        // retrieve reference to CompAResource from CompAResourceHome.
    }

    public CreateResourceResponse createResource(CreateResource req)
    throws RemoteException
    {
        // create new endpoint reference and return it
    }

    public DoActionResponse doAction(DoAction req) throws RemoteException
    {
        CompAResource resource = getResource();
        resource.doActivity1(req.getParameters());
        resource.doActivity2(...);
        ...
    }
}
```

The Figure 1.1 and Figure 1.2 explain the relationship between a resource, a resource home, and a service implementation. For better illustration of the service implementation, Listing 5.1 shows the pseudo codes of a common component service implementation. A general component service implementation extends the abstract base class `PrimitiveComponent` (see Figure 5.6) and implements two methods `getResource()` and `createResource()`. The method `createResource()` is always the first method to invoke by a web service client to instantiate a component resource and to get the endpoint reference of the resource (the combination of the service URL and the resource id). All the subsequent in-

vocations to grid methods like **doAction()** will get a resource reference by calling **getResource()** method, and invoke a series of methods of the resource.

The methods in **PrimitiveComponent** are divided into two categories. One is for interactions with the organizer, as listed in Listing 5.2. The other category is for interactions with its peer component services, as listed in Listing 5.3.

Listing 5.2 Interfaces for the organizer

```
createResource();  
initComponent();  
subscribeStatus();  
start();  
getStatus();  
stop();
```

Listing 5.3 Interfaces for peer component services

```
canReceiveEntity();  
receiveEntity();  
receiveVariableUpdate();  
getSharedVariable();  
updateTimeAdvancement();  
unsubscribe();
```

In the first category, the **createResource()** method will be invoked by the organizer to instantiate component resources. The return of the **createResource()** is an endpoint reference including both the service URL and the resource id which identifies the new resource created. The **initComponent()** method is invoked by the organizer to initialize a component resource instance. The organizer constructs an **InitComponent** object to encapsulate all the initialization information for each component resource instance. As shown in Figure 5.6, the **InitComponent** encapsulates the information about **ComponentInstance** (the setup variables and the operation rule configurations - see **componentInstances** in Figure 4.3), and **LinkedComponents** (the preceding components, the following components, the provider components, the consumer components, and the endpoint references of all these components). If the organizer invokes the **subscribeStatus()** of a com-

ponent, the organizer will be notified when the component resource finishes execution. The organizer invokes the **start()** method to start the simulation. During the simulation execution, the organizer can check the status of the component resource by invoking the grid method **getStatus()** of the component. The organizer invokes **stop()** method of each component to terminate the simulation prematurely when required.

The other category of methods is for interactions with its peer component resources. These grid methods enable communication between component resources in a purely decentralized manner. During a simulation execution, component resources interact with each other by transferring entities, exchanging shared variables, and sending time update acknowledgements. These are facilitated by the grid methods **canReceiveEntity()**, **receiveEntity()**, **getSharedVariable()**, **receiveVariableUpdate()**, and **updateTimeAdvancement()**. Method **canReceiveEntity()** is invoked by a preceding component resource to check whether current component resource can receive an entity. If it returns true, the preceding component resource will invoke the **receiveEntity()** method of the current component to transfer an entity. When a consumer component instance needs to retrieve the shared variable value, it invokes the **getSharedVariable()** method of the current component resource. If the shared variable value is not available, the consumer component resource will get a return with null value and the main thread (primitive component manager thread) will wait. Once the shared variable value is available, the consumer component's **receiveVariableUpdate()** method will be invoked to receive the updated shared variable value and the main thread will be waked up. When the simulation time of a component resource is updated, the component resource will invoke the **updateTimeAdvancement()** method of all its constrained components.

Figure 5.6 Class diagram of PrimitiveComponent

Method **getResource()** returns the resource instance to be operated on by other methods, as illustrated in Listing 5.1. Notice that there is no **subscribe()** method for peer component services. This is because the subscription of shared variables are carried out automatically inside the **initComponent()** method when the organize initialize each component resource. If a component resource finishes execution, it should explicitly unsubscribe its subscribed variables by invoking the **unsubscribe()** method of relevant the provider component resources so that the provider component resources do not send variable update in the future.

The detailed descriptions of the methods in both categories are presented in Appendix F.1.

5.2 Facilities for Component Development

In the component-based view, SOAr-DSGrid provides a set of schemas for component development, including entity type file, component interface file, and composite component implementation file. SOAr-DSGrid provides facilities to transform the XML files in the component-based view into invocable and executable component services (WSRF Web Services) in the service-oriented view. SOAr-DSGrid includes a module called *service generator* to realize this functionality. The basic idea is to use XSLT engine to generate source files for component services by combining the information in XML files in component-based view and the templates predefined in the service generator, and then build and deploy the component services into a GT4 container. This section describes the service generator in detail.

5.2.1 General Code Generation

Generally, code generators relieve developers from writing code manually. The code generation process is repeatable and trackable. This leads to consistency of the quality of the code generated. In case of any bug in the generated code, the bug can be fixed at the tem-

plate level and the code can be regenerated.

Figure 5.7 General code generation model

Figure 5.7 depicts the general code generation model. A design serves as the variable input to the code generator. A template is the abstraction of the common logic embedded in the codes generated. A code generator applies certain design to a template, and generates code accordingly. A code generator is necessary only when there are multiple designs exist for a single template. A template can be an XSL template or a UML model.

5.2.2 Service Generator

The core of the SOAr-DSGrid service generator is a class `ServiceGeneratorEngine`. The `ServiceGeneratorEngine` uses the `SchemaValidator` to validate schemas and uses the `CodeGenerator` to generate component service codes. Figure 5.8 depicts the class diagrams of `ServiceGeneratorEngine`, `CodeGenerator`, and `SchemaValidator`.

A component interface file contains the description about the component type, the variables, and the operations of the component. However, as the component interface file is created in the component-based view, it does not contain any information about the implementation and deployment of the component service. For the same component interface file, it can be used to generate a component service with any package name, class name, namespace, and gar file name. `ServiceGeneratorEngine` can use default values for implementation and deployment properties. In order to provide flexibility in component service generation and deployment, SOAr-DSGrid enables a component provider to specify an XML file, namely `serviceGenXML`, to provide the preferred implementation and deployment information about the component service. However, for easier service dis-

covery, the service name in GT4 container is the same as the corresponding component type specified in the component interface file and the simulation description file. The resource instance is the same as the combination of the session name and the name of the component instance in the simulation description file. The combined information of component interface and serviceGenXML will be used six times as shown in Figure 5.9. In order to improve the efficiency of the service generation, SOAr-DSGrid creates an in-memory intermediate XML and uses it six times instead of combining the information six times. The component provider can optionally configure the service generator to write the intermediate XML to a physical file for debugging purpose. Therefore, in the ServiceGeneratorEngine class diagram, there are attributes compIntf, serviceGen, and intermediateXMLContent.

Figure 5.8 Class diagram of ServiceGeneratorEngine

When generating a component service, the ServiceGeneratorEngine first uses the SchemaValidator to validate the component interface file and serviceGen XML against the respective schemas. Upon successful validation, it will invoke the code generator to generate component service files one by one. The complete service generation and de-

ployment process is presented in Figure 5.9.

Figure 5.9 Component service generation process

As shown in Figure 5.9, the component interface XML file and ServiceGen XML file will be sent to the service generator engine to generate an intermediate XML. The intermediate XML will then be sent to the code generator subsequently. The essence of the code generator is an XSLT engine and it follows the general code generation process depicted in Figure 5.7. In SOAr-DSGrid, the “design” is the intermediate XML, and the “template” is an XSL template. As described in section 5.1, SOAr-DSGrid follows the WSRF programming model. Therefore, a template is created for each of the following: the component resource, the component resource home, the component service implementation, the WSDL, the server deployment descriptors, and the JNDI configuration. The component resource, the component resource home, and the component service implementation are the three key files elaborated in section 5.1. The generated component resource class extends the base abstract class `PrimitiveComponent`. The definitions of the component-specific variables and their corresponding accessor methods are included in the generated component resource class. In order to generate a fully deployable and invocable GT4 Web Service, some server-side configuration information is also required.

The WSDL is for the description of the GT4 Web Service. The server deployment descriptor contains information such as the service name in GT4 container, the class name of the component service, whether to load the web service when the GT4 container is started or only to load it on first invocation of the web service, and etc. The JNDI configuration contains information for component resource lookup such as the resource home class name, the resource class name, the resource key type and name, and etc. The code generator applies the intermediate XML to different templates to generate different files. Six XSL templates are created for the generation of a component service in GT4, i.e., `ComponentResource.xml`, `ComponentResourceHome.xml`, `Component.xml`, `server-config.xml`, `jndi-config.xml`, and `wsdl.xml`. All the six templates are provided as part of SOAr-DSGrid and are common to all component services. For `ComponentResource.xml`, `ComponentResourceHome.xml`, and `Component.xml`, they are defined by extending respective base classes elaborated in section 5.1. All the templates are listed in Appendix C.1 to Appendix C.6 for reference.

In the component interface file, the component provider specifies the fully qualified class names of the operations supported by the component. This information about the operations is captured in the generated component resource. However, the actual operation class, which is provided by the component provider in a separate jar file, is not required to be ready when generating the component service. It is neither required during the deployment of the generated component service, as it is not part of the deployment file. The primitive component manager tries to load the operation class from the classpath when the component resource starts simulation. Therefore, as long as the operation class is in the classpath at the time a component resource starts simulation, there will be no error. This lazy operation-loading feature effectively separates the component service generation and the operation development. The component provider can concentrate on the development of the operation, i.e., the simulation logic, without worrying about the cou-

pling between the component service and the operation.

Besides the ServiceGeneratorEngine API for service generation, SOAr-DSGrid also provides a simple GUI to the component provider to generate, build, deploy, undeploy, and register a component service. This GUI also enables the component provider to start and stop the GT4 container, as the registration of a component service requires the index service has started in the GT4 container. Appendix E.1 shows the snapshot of the GUI.

5.3 Shared Variable Management

In pure component-based software engineering, each component is independent of other components, and the components are loosely coupled. However, in component-based simulation, sharing variables between simulation components are common. The support for shared variable management in SOAr-DSGrid is developed with this concern. The variable sharing in the service-oriented view must also ensure the loose coupling between component services. SOAr-DSGrid implements shared variable management using the receive order (RO) messages instead of time stamp order (TSO) messages. This section details the shared variable management in SOAr-DSGrid.

5.3.1 Data Structures and Web Service Interfaces

At the component design time, each component bears a component interface conforming to a schema. At runtime, each component in SOAr-DSGrid is a WSRF Web Service. SOAr-DSGrid provides grid methods for retrieval of shared variables. Figure 5.10 shows the class diagram of the relevant classes involved in shared variable management. Unlike Object-Oriented Architecture (OOA), Service-Oriented Architecture (SOA) separates the data from the operations. In SOAr-DSGrid, all the grid methods are encapsulated in the web service implementation class PrimitiveComponent, and all the data are encapsulated in the PrimitiveComponentResource. The execution of each grid method starts with the retrieval of a resource instance, followed by invocation of a corresponding method of the

resource instance.

The following main data structures are used in the implementation (see Figure 5.10):

- **TimedRecord**, which is a tuple of (value, validFrom, validTill): value is the object representing the value of the shared variable, validFrom is the update time from which the value is valid, and validTill is the time from which the current value becomes invalid. When a shared variable is updated, the validTill field of the latest record will be updated with the current simulation time. A new record with both validFrom and validTill equal to the current simulation time will then be created.
- **HistoryList**, which is the implementation of the history list in the owner. The HistoryList maintains a list of TimedRecord objects and accessor methods to the list.
- **SharedVariable**, which is the class representation of a shared variable. It encapsulates a history list object. It also contains the subscriber list (i.e., a list of components that subscribe to the shared variable).
- **FutureList**, which is the implementation of future list in the reader. Similar to the history list, it contains a list of TimedRecord objects.
- **SubscribedVariable**, which is the class representation of a subscribed variable. A SubscribedVariable, the local shadow copy of the SharedVariable, is maintained by the reader. The SubscribedVariable encapsulates a future list object.

Each component instance has an instance of PrimitiveComponentResource. The owner component creates a SharedVariable object for each shared variable, and puts this object into the sharedVariables HashMap in the PrimitiveComponentResource instance. A SubscribedVariable object is used by the reader to maintain a local copy of each remote SharedVariable object. The created SubscribedVariable object is put in the reader's subscribedVariables HashMap. The HashMap sharedVariables and subscribedVariables are used to map the local variables defined in PrimitiveComponentResource to the SharedVariable object or SubscribedVariable object respectively.

Figure 5.10 Data structures and interface for shared variables

As elaborated in subsection 5.1.1, SOAr-DSGrid defines an abstract class named *Operation*. A subclass of *Operation* contains the application specific simulation logic. The execution of an *Operation* may use a local variable corresponding to the shared variable of another component instance. The *Operation* retrieves the value required by invoking the *getSubscribedVariable()* method of *PrimitiveComponentResource*. This method calls *getRecord()* method defined in the *SubscribedVariable* class, which in turn tries to get the value from the future list of the *SubscribedVariable* object. As for an owner, the *Operation* may modify the values of some shared variables. *synchronizeSharedVariables()* is used by the owner to add a record to the *SharedVariable* object using the value of the local variable which corresponds to the shared variable.

In the case that the reader's future list does not contain the required value, the reader will use the grid method **getSharedVariable()** to get the updated values of the shared variable from the owner. When the owner updates its simulation time, the Notifier of the owner will invoke the corresponding reader's **receiveVariableUpdate()** grid method: ei-

ther push the latest record to the reader in an unsolicited manner or return the record to the reader as the response of an unfulfilled request. At the reader side, the Notification-Processor will save the received updated record into the future list if there is any.

Figure 5.11 depicts the sequence of retrieval, update, and fossil-collection of shared variables. How these activities are used in the basic Pull without FutureList, Pull with FutureList, and Push mechanisms is described in the next subsection.

5.3.2 Basic Mechanisms

The underlying mechanisms described in this section are similar to the methods proposed in [54]. The Pull without FutureList is the same as PullROTG and the Push method is similar to PushROTG. Push is different from PushROTG when the reader runs ahead of the owner. For PushROTG, the reader will issue a pull request to the owner if the reader runs ahead of the owner. But for Push, the reader will not send a pull request to the owner and will wait till the requested value is pushed by the owner. The author introduced a new approach, Pull with FutureList to improve the performance of PullROTG.

Pull Without FutureList

The Pull without FutureList is the same as the PullROTG in [54]. In this approach, the owner maintains a history list for each shared variable, and the reader maintains a local cache of the last received TimedRecord (not a FutureList) for each subscribed variable. When the reader requires the value of the shared variable at time t , it first checks whether the requested time falls in the validity range of the local cache. If the local cache satisfies the condition $validFrom \leq t < validTill$ or $validFrom = t = validTill$, then the *value* will be returned immediately to the main execution thread of the component. Otherwise, the local cache is considered as obsolete. A remote invocation of `getSharedVariable()` will be issued to the owner.

Figure 5.11 Sequence diagram for shared variable update

When the owner receives the request, it will try to get the requested record from the history list. The owner will return the single record which satisfies the condition *validFrom* $\leq t < \text{validTill}$ or *validFrom* = *t* = *validTill* as a response to the reader. If no such record is found, the owner will add the request to the unfulfilled list, and return a null record to the reader immediately without blocking the invocation. When the owner advances simulation time and the requested record becomes available, the Notifier of the owner will invoke the **receiveVariableUpdate()** grid method of the reader.

When the reader receives the response, the NotificationProcessor of the reader will save the record as the fresh cache copy and return the value to the main execution thread if the response is not null. Otherwise, the reader's main execution thread will keep waiting until it is notified by the NotificationProcessor after the NotificationProcessor receives the record from the owner.

Pull With FutureList

The Pull with FutureList is similar to the Pull without FutureList. The differences are:

- When the owner receives a **getSharedVariable()** invocation, instead of returning a single record, it will check the history list to find all the records which satisfy the condition *t* < *validTill* or *validFrom* = *t* = *validTill*.
- Since the reader maintains a future list, it will only send a pull request to the owner if it cannot find the requested value in the future list. The reader also needs to fossil-collect the future list properly.

Push

In the Push approach, the reader maintains a future list for each subscribed variable. This approach is based on the always-update-by-writer update policy, similar to the one used by Lim et al. [52]. The Push is similar to PushROTG, but it is different from PushROTG when the reader runs ahead of the owner.

When the reader requires a variable, it always tries to retrieve the value from the corresponding future list first. If in the future list there is a record that satisfies the condition $validFrom \leq t < validTill$ or $validFrom = t = validTill$, then the *value* will be returned immediately to the main execution thread of the component, and the future list will be fossil-collected accordingly. Otherwise, the reader's main execution thread will wait till the requested record is received from the owner through the reader's NotificationProcessor after it received the required record from the owner. This is different in PushROTG. In PushROTG, the reader will send a pull request to the owner if the reader runs ahead of the owner.

At the owner side, if a variable update happens, the Notifier thread will push the updated record to each reader through the **receiveVariableUpdate()** grid method of the reader. As described above, when the reader receives the notification, the NotificationProcessor will add the updated record to the future list, and subsequently notify the main execution thread of the component to read from the future list.

Fossil-Collection

In all approaches proposed and implemented, a history list and/or a future list are involved. If the sizes of these lists keep on increasing, the cost of searching and the memory usage will also increase, and it will finally reach a point where the solution becomes infeasible. Thus, an effective and proper fossil-collection is required for both the history list and the future list.

Fossil-collection happens whenever there are some obsolete records in the lists. As for the history list, when the global minimal simulation time of all the subscribers increases, the history list should be fossil-collected. As for the future list, whenever an Operation requests for a subscribed variable at a certain time, all the records whose *validTill* is smaller than the request time are considered as obsolete. However, the latest record in the history

list should never be fossil-collected⁴. The record whose *validTill* is equal to the request time should not be fossil collected as this record may be requested again with the same request time. Listing 5.4 and Listing 5.5 show the algorithms used for fossil-collection of the history list and future list respectively.

Listing 5.4 Fossil-collection of the history list

```

1 removeTill // remove all records with validTill <
  removeTill (except for the last record)
2 while(there is a record) {
3   if(record is the latest record) break;
4   else if(record.validTill < removeTill){
5     remove this record;
6   } else { break; }
7 }
```

Listing 5.5 Fossil-collection of the future list

```

1 removeTill // remove all records with validTill <
  removeTill
2 while(there is a record) {
3   if(record.validTill < removeTill)
4     remove this record;
5   } else { break; }
6 }
```

5.4 Experiments

The experiments are divided into two parts. In the first part, two sample component services are generated and the basic profiling is carried out. In the second part, the performances of different approaches in shared variable management are compared. The two component services deployed in the first part are used for the shared variables experiment.

5.4.1 Component Service Generation and Profiling

Component Services Development

In this part, two SOAr-DSGrid components are developed using the service generator GUI provided by SOAr-DSGrid. The first component CompA shares a variable value to other components. It is the owner of the shared variable and it updates the shared variable

⁴ *validFrom* and *validTill* fields of this record are equal and the *validTill* field will be set to the updated simulation time when the next time the shared variable is updated.

value, and thus it is referred to as an *owner component*. The second component CompB reads the value of the shared variable of CompA, and thus it is referred to as a *reader component*. The generation processes of the two components are the same. Therefore, only the service generation and profiling of the owner component alone are described.

The component interface file is defined as CompA.xml (refer to Appendix B.2). There are three variables: a string variable *var1*, two int variables *var2* and *var3*. The variable *var3* is a shared variable. The operation supported is *soardsgrid.operation.OperationA*. In order to customize the package names and GT4 service names of the generated CompA component service, a serviceGen XML is also defined (refer to Appendix B.4). The service generator GUI as shown in Appendix E.1 is used to generate the component service source code. The GUI is also used to build and deploy the CompA component service. After that, the GT4 container can be started from the GUI and the CompA can be registered with the index service through the GUI. The same process is repeated for the generation of CompB component service.

The reader component supports an operation *soardsgrid.operation.OperationB*. Before the execution of the two components, a jar file which contains the class *soardsgrid.operation.OperationA* (refer to Appendix B.1) and another jar file (or the same jar file) containing *soardsgrid.operation.OperationB* must be put on the classpath of the GT4 container. However, These classes do not need to be present when generating the component service source, building the service build, deploying the service, or registering the service. The two operations are illustrated in subsection 5.4.2.

Profiling

The CompA class only defines `createResource()`, `getResource()`, and some accessor methods for variables *var1*, *var2*, and *var3*. The basic profiling is carried out on the **createResource()** grid method and `getResource()` method. A client program is developed to invoke the `createResource()` grid method. As shown in Figure 5.12, the round trip of an invoca-

tion to a grid method involves seven subprocesses: the serialization of the request, the network transmission of the request, the deserialization of the request, the server processing, the serialization of the response, the network transmission of the response, and the deserialization of the response.

Figure 5.12 Round trip of a grid invocation

The client program records the total round trip time. The profiler tracks the server processing time and the number of invocations. The client overhead includes all the subprocesses except for the server processing. The profiling is carried out using NetBeans IDE 6.5 with 32 runs. The results are shown in Table 5.1.

The `getResource()` method is a local method in the CompA service. It takes around 0.59 milliseconds for a single invocation. The client create resource time in the table header refers to the round trip time at the client side. The server create resource time in the table header refers to the time spent for the server processing. From the table, it can be seen that the time for `createResource()` at the server side is minimal. It takes around 6.8 milliseconds and this contributes only 6.79% of the total create resource time. Therefore, the improvement on the performance of the `createResource()` grid method will highly rely on the performance improvement of the SOAP binding (the conversions between Java and XML) and the network transmission, which are out of scope of this thesis.

From the table, it also can be seen that the first grid invocation takes around four times of s subsequent invocation. As for the create resource at the server side, the first invocation takes 226 milliseconds, which is much greater than the time on s subsequent invoca-

tion. This is because the first invocation involves the initializations of the resource home and the first TCP connection. All the subsequent invocations will reuse the established persistent TCP connection.

Table 5.1 createResource getResource profiling results

Runs	client create resource time (ms)	server create resource time (ms)	client overhead (ms)	server get sesource time (ms)
1	440	226	214	0.726
2	110	5	105	0.654
3	100	5	95	0.56
4	111	6	105	13.16
5	100	4	96	0.5
6	90	6	84	0.5
7	100	3	97	0.6
8	101	4	97	0.5
9	90	6	84	0.4
10	101	12	89	0.5
11	100	5	95	0.5
12	110	7	103	2.1
13	100	9	91	0.4
14	90	5	85	0.5
15	110	10	100	0.5
16	130	7	123	0.6
17	100	7	93	0.5
18	100	9	91	0.5
19	90	6	84	0.4
20	100	7	93	0.5
21	90	5	85	0.5
22	100	6	94	0.4
23	100	8	92	0.5
24	90	8	82	0.9
25	100	7	93	0.4
26	100	10	90	0.4
27	91	7	84	0.5
28	100	6	94	0.5
29	100	8	92	0.4
30	100	5	95	0.5
31	110	10	100	1.4
32	100	7	93	0.5
Average⁵	100.10	6.80	93.30	0.59
Percentage⁶	100%	6.79%	93.21%	

⁵ The average is calculated excluding the result of the first invocation. It also excludes the abnormal result of the 4th run.

⁶ The percentage is the subprocess over the total client createResource time, e.g., $6.80/100.1 \times 100\% = 6.79\%$. The server getResource is not included in the client createResource process, and thus there is no percentage value for server getResource.

5.4.2 Shared Variable Experiments

In this part, two experiments are carried out to compare the performance of the three approaches in shared variable management: Pull without FutureList, Pull with FutureList, and Push. The same simulation model is used in the experiments with different configurations. The simulation terminates at a specific simulation time for all experiments. In the simulation model, there are two components. One is the owner component CompA with OperationA. The other one is the reader component CompB with OperationB. Both components are automatically generated and deployed into GT4 container as described in subsection 5.4.1.

After OperationA is triggered, the owner will do the following:

- 1) Executes a spin-loop for a period, defined by OwnerUnitProcessingTime;
- 2) Updates the value of the shared variable and synchronizes the shared variable; and
- 3) Advances its simulation time by an amount defined by UpdatePattern.

After OperationB is triggered, the reader will do the following:

- 1) Advances its simulation time by an amount defined by RequestPattern;
- 2) Retrieves the shared variable; and
- 3) Executes a spin-loop for a period defined by ReaderUnitProcessingTime.

The following Table 5.2 summarizes the factors that affect the simulation performance.

Table 5.2 Simulation factors

Factor	Description
UpdatePattern	The update interval (in simulation time) of the owner
RequestPattern	The request interval (in simulation time) of the reader
Owner Unit ProcessingTime	The emulated processing time (in wall-clock time) between two consecutive simulation time advancements of the owner
Reader Unit ProcessingTime	The emulated processing time (in wall-clock time) between two consecutive simulation time advancements of the reader

In the first experiment, we vary the unit processing time of the reader. In the second experiment, we vary the request pattern of the reader.

Varying Unit Processing Time

In this experiment, the configuration of the simulation is summarized in Table 5.3.

From Figure 5.13, it can be seen that when the reader unit processing time is smaller than the owner unit processing time, the Push gives the best performance. Since the reader unit processing time is smaller, the reader runs ahead of the owner. In this case, for both Pull without FutureList and Pull with FutureList, the reader will issue a pull request through the **getSharedVariable()** grid invocation to the owner. The owner will return a null record immediately because the requested record is not in the HistoryList of the owner. This will cause the reader to wait until the owner's simulation time advances to the requested time. Then, the owner's Notifier will return the requested record to the reader through the **receiveVariableUpdate()** grid invocation. Thus, either in Pull with or without FutureList, these **getSharedVariable()** grid invocations are essentially redundant. In the Push approach, these redundant invocations are avoided and consequently the performance of Push is better than the other two.

Table 5.3 Simulation configuration (varying reader unit processing time)

Paramter	Value
UpdatePattern (simulation time)	10
RequestPatten (simulation time)	10
OwnerUnitProcessinTime (wall-clock time in millisecond)	50
ReaderUnitProcessingTime (wall-clock time in millisecond)	3, 5, 10, 20, 40, 50, 60, 80, 100, 200

The experiment results are summarized in Figure 5.13. The figure also shows that the Pull without FutureList and Pull with FutureList have similar performance when the reader runs ahead of the owner. It is clear that the Pull with FutureList will degrade to the Pull without FutureList as there is always zero record returned to the reader for every **getSharedVariable()** grid invocation.

Figure 5.13 Experiment results (varying reader unit processing time)

When the reader unit processing time is greater than the owner unit processing time, the owner runs ahead of the reader. In this case, the Pull with FutureList gives the best performance. Since the owner runs ahead of the reader, multiple records may be available to the reader when the reader issues the `getSharedVariable()` grid invocation. So at the reader side, most of the requests of the values of the shared variable can be fulfilled by the FutureList. As for Push, most of the requests can also be fulfilled from the FutureList. However, since the owner only pushes one record to the reader each time, the number of `receiveVariableUpdate()` grid invocations will be similar to the number of grid invocations in Pull without FutureList. That is why Push has a similar performance as that of Pull without FutureList when the owner runs ahead of the reader.

Varying Request Pattern

In this experiment, the configuration of the simulation is summarized in Table 5.4. The experiment results are summarized in Figure 5.14.

Table 5.4 Simulation configuration (varying request pattern)

Parameter	Value
UpdatePattern (simulation time)	10
RequestPatten (simulation time)	2, 4, 8, 16, 32, 64, and 128
OwnerUnitProcessinTime (wall-clock time in millisecond)	50
ReaderUnitProcessingTime (wall-clock time in millisecond)	50

In this experiment, the owner and the reader have the same unit processing time. The relative relationship between the update pattern and the request pattern will determine whether the owner runs ahead of the reader or the other way.

Figure 5.14 Experiment results (varying request pattern)

When the request pattern is less than update pattern, the owner essentially runs ahead of the reader since the owner's simulation time advances more than the reader's in the same wall-clock time. In this case, the Pull with FutureList gives the best performance. As the update pattern is larger, when the reader requests for the value, a single record's validity period will be greater than the request pattern. The same record will be used to fulfill several subsequent requests. During this period, the owner advances its simulation time further. Therefore, the next `getSharedVariable()` grid invocation will fetch more re-

cords, and more requests will be fulfilled by the FutureList. In total, the number of **getSharedVariable()** grid invocations in Pull with FutureList is much smaller than that of the Pull without FutureList. The number of **receiveVariableUpdate()** grid invocations in Push is fixed since each update will lead to one **receiveVariableUpdate()** grid invocation.

When the request pattern is greater than the update pattern, the reader advances simulation time faster. As the request pattern is greater, in both Pull without FutureList and Pull with FutureList, the reader needs to issue a **getSharedVariable()** grid invocation and wait until the owner's Notifier issues a **receiveVariableUpdate()** grid invocation. The total numbers of grid invocations for Pull without FutureList and Pull with Future list are the same. However, for Push, the total number of **receiveVariableUpdate()** grid invocations is larger than the total number of grid invocations in Pull with FutureList, and most of the pushed records are not used by the reader since the request pattern is greater than the update pattern. Therefore, the Pull with FutureList and Pull without FutureList have similar performance, and the Push has the worst performance.

5.5 Summary

This chapter begins with the description of what a component service is in service-oriented view. The SOAr-DSGrid component services strictly follow the programming model of WSRF. The component resource contains all the elements required for component execution. These elements include the external entity queue, the internal entity queue, the output entity queue, the primitive component manager, the output queue manager, the local rule engine, the local time manager, the shared variable notifier, and the shared variable notification processor. The interactions between component resources, e.g., the transfer of entities, the queries and notification about shared variable values, and the update of time advancement, are all carried out through the web service interfaces exposed by the component service implementation. SOAr-DSGrid provides a service generator to trans-

form a component interface file in the component-based view into a component service in service-oriented view. The service generation relieves the component provider from writing component services manually. It improves the productivity of the component provider, the quality and the consistency of the generated component services.

Three approaches namely, Pull without FutureList, Pull with FutureList, and Push, are implemented for shared variable management. Pull with FutureList is a greedy version of Pull without FutureList. The Pull with FutureList generally has better or the same performance as the Pull without FutureList. The speed of the simulation is affected by four factors in two dimensions. The first dimension is the simulation time advancement amount (i.e., update pattern and request pattern), and the second is the unit processing time (i.e., the owner unit processing time and the reader unit processing time). If the reader runs ahead of the owner because the reader unit processing time is smaller, then the Push has better performance than the Pull with FutureList. For all the other three combinations, the Pull with FutureList has better performance over the Push.

CHAPTER 6

SIMULATION LIFECYCLE

In Chapter 4, we presented the component-based view on component development and simulation application development. Chapter 5 explains the bridge between the component-based view and the service-oriented view for component development. The service generator transforms component interface files in the component-based view into component services in the service-oriented view for component providers. This chapter will explain the bridge between the component-based view and the service-oriented view for simulation process. SOAr-DSGrid provides facilities for running a simulation for the clients. The organizer receives the simulation description file, locates component services, and launches a runtime simulation. The component services realize the distributed simulation execution. The following sections detail the initialization phase, execution phase, and termination phase of the simulation process lifecycle.

6.1 Simulation Initialization

A client initiates a simulation by sending a simulation description file to the organizer. The organizer is responsible for transforming the simulation description file into a simulation execution. The organizer initializes and triggers the simulation execution. The sequence diagram of the simulation initialization is shown in Figure 3.7 and the details are presented under subsection 3.2.3 of this thesis. This section will discuss the runtime de-

tails of the organizer and the roles of the organizer during the initialization of a simulation.

Functional-wide, the organizer acts as an agent of the client. It processes the simulation description file, transforms it into a simulation exeuction, and triggers the component services on behalf of the client. At the implementation level, the organizer is implemented as a WSRF Web Service. We use the term *organizer service* in the service-oriented view to differentiate it from the organizer in the component-based view. To start, stop, and monitor the simulation execution, the organizer service should expose some grid methods to the client. Figure 6.1 depicts the class diagrams of the organizer service and the organizer resource. The relationships between a resource, a resource home, and a service implementation have been discussed in section 5.1.

Figure 6.1 Class diagrams of the organizer service and organizer resource

In Figure 6.1, the client will first invoke the **createResource()** grid method to generate a new organizer resource. The client then invokes the **submit()** grid method of the organizer service. Upon receiving this invocation, the organizer service will store the original simulation description file into a staging directory of the server that hosts the organizer

service. The location of the simulation file is saved in the *simFilePath* variable of the organizer resource instance. The **start()** grid method will invoke the **start()** method of the organizer resource which actually triggers the initialization process of a simulation. During the simulation execution, the organizer does not coordinate among the component resource instances. However, the organizer might stop the simulation execution or query the status of a component resource instance upon requests from the client.

Figure 6.2 Initialization process of a simulation

The **start()** method of the organizer resource actually maps the simulation description file in the component-based view into a simulation execution in the service-oriented view. Figure 6.2 shows the flow of this process.

Firstly, the organizer invokes an XML parser to parse the simulation description file into a DOM tree. In order to minimize parsing failures, the simulation description file is validated against the corresponding XML schema before parsing. The schema is attached in Appendix A.4. The actual parsing steps comprise parsing the component resource instances, parsing the rule configurations, parsing the entity flows, parsing the variable flows, and parsing the results.

Secondly, the parser transforms the DOM trees into a *Simulation* object, an internal object used by the organizer. The *Simulation* is a class representation of the simulation schema. It contains a variable *sessionName*, *ComponentInstance* objects, a *Composition* object, and a *Result* object (refer to Figure 4.7 and *ComponentInstance* in Figure 5.6). We

name the set of these objects and variables as a simulation session. The transformation process extracts the topology information in the Composition object (presented in the form of entity links and variable links), and creates the LinkedComponents object shown in Figure 5.6, including the preceding/regulating component resources, the following/constrained component resources, the provider component resources, and the consumer/subscriber component resources for each component resource. The LinkedComponents object will be used as a parameter to invoke **initComponent()** grid method of a component service.

Thirdly, the organizer consults the index service to find the URLs of the involved component services according to the component types specified in the simulation description file.

Then, the organizer creates component resource instances according to the component names and types specified in the simulation description file. The organizer resource will maintain the resource references of all the created component resource instances.

After that, the organizer resource initializes each component resource. This includes passing a component resource's topology information (LinkedComponents object generated in the second step), i.e., its preceding/regulating component resources, its following/constrained component resources, its provider component resources, and its subscriber component resources. Other information like the references to these respective component resources, the setup variables, and the rule configurations is also passed to the component resource.

Then, the organizer subscribes itself with the individual component resources for their status. This step will ensure proper termination of a simulation. It will be further discussed in section 6.3 under simulation termination.

Lastly, the organizer invokes the **start()** grid method of each component service to trigger the simulation execution of each component resource.

During the simulation initialization, the organizer service needs to query the index service to find the URL of a component service before creating the corresponding component resource. After successfully creating a component resource, the organizer can optionally register the component resource with the index service.

6.2 Simulation Execution

When the organizer service invokes the **start()** grid method of the component service, the component service thread⁷ invokes the start() method of a particular component resource. The component resource then starts the simulation execution. Figure 6.3 shows the flowchart of the start process and the simulation execution.

Figure 6.3 Flowchart of simulation start in component resource

Four long running threads, namely, the output queue manager, the shared variable notification processor, the shared variable notifier, and the primitive component manager are started sequentially. Upon the successful start of the four threads, the status of the component resource will be changed to “*running*”. The four threads will coordinate among each other to continue the simulation without any interference of the organizer. The component service thread may interfere with the four threads intermittently during the simulation execution, e.g., when **updateTimeAdvancement()** grid method is invoked, the compo-

⁷ The GT4 container creates a component service thread when the component service is started. The GT4 container is responsible for the thread pooling and the termination of a component service thread.

nent service thread will wake up the primitive component manager thread (refer to the description on the time manager in subsection 5.1.1). The four threads will stop until the status of the component resource is changed to “*terminated*”.

The primitive component manager is the coordinator among all the active threads and the passive objects. The primitive component manager thread is considered as the main simulation thread. It assigns tasks to the shared variable notifier thread and the output queue manager thread, and then notifies them accordingly. It also invokes the passive objects, i.e., the time manager, the rule engine, and the operations. Refer to Figure 5.4 for the detailed flow of the primitive component manager.

The output queue manager will be in wait state if there is no entity in the output entity queue. It will be notified by the primitive component manager thread when the primitive component manager moves the processed entities or the generated entities to the output entity queue after successful operation execution.

The shared variable notification processor thread is in wait state if there is no remote notification task. The component service thread will assign a notification task, i.e., a shared variable update event, to the shared variable notification processor thread when the component service thread receives an invocation on the **receiveVariableUpdate()** grid method. The component service thread will also notify the shared variable notification processor thread to process the notification task.

The shared variable notifier thread is used by a writer component resource to notify its consumer/subscriber component resources. It remains in wait state if there is no update on the shared variables. If the operation in the primitive component manager thread updates the value of a shared variable and the time manager has executed the corresponding internal event on the variable update, a shared variable update event will be generated and passed to the shared variable notifier thread. The shared variable notifier thread will send the events to the consumer/subscriber component resources via invocation on the grid

method **receiveVariableUpdate()** of the consumer/subscriber component service.

During the simulation execution, the component service thread could receive two kinds of requests from the organizer service, i.e., querying the status of a particular component resource and terminating the simulation prematurely.

The whole simulation execution is distributed and self-coordinated. There is no intermediary between peer component services for control flow or data flow. This relieves the overhead of centralized control flow and data flow.

6.3 Simulation Termination

In component-based view, the termination of a simulation involves the return of simulation results if any. In the service-oriented view, the termination involves not only returning simulation results, but also the release of computational resources.

As explained in section 6.2, SOAr-DSGrid uses four active threads and a component service thread for simulation execution. The GT4 container creates the component service thread, and it is responsible for the thread pooling and the termination of the component service thread. The four active threads are created by the component resource upon the invocation of the `start()` method. Therefore, SOAr-DSGrid runtime is responsible for the termination of these four threads to avoid computation resource leakage. Thus, certain mechanism must be in place to ensure the proper and successful termination of a simulation.

In SOAr-DSGrid, a simulation could be terminated in two ways. First, a simulation can be aborted by the client and the simulation is terminated prematurely. Second, a simulation is executed until a termination condition (e.g., the simulation time of the component resource reaches a predefined value) is satisfied. In both cases, the status of the component resource will be changed to “*terminated*”. Figure 6.4 and Figure 6.5 illustrate the sequences of premature simulation termination and normal simulation termination respec-

tively.

In a prematured termination, the client gives instructions to abort the simulation, and the organizer invokes the **stop()** grid methods of each component service. Subsequently, the **stop()** method of the component resource (refer to as **CompAResource** in Figure 6.4 for an example) will be invoked by the component service implementation (refer to as **CompA** in Figure 6.4 for an example). The **stop()** method of component resource involves invocations on **terminate()** method of all the four active threads, i.e., the primitive component manager, the output queue manager, the shared variable notification process, and the shared variable notifier. The component resource also sets a termination time for itself. The GT4 container will destroy the component resource and release the relevant computing resources. Similar to the component resource, the organizer resource will also set immediate termination for itself, and the GT4 container will destroy the organizer resource.

The primitive component manager initiates a normal termination by invoking the **terminate()** method of its component resource (e.g., **CompAResource** in Figure 6.5) once certain termination condition is satisfied. The component resource (e.g., **CompAResource** in Figure 6.5) unsubscribes itself to its provider component services (e.g., **CompB** in Figure 6.5) by invoking the **unsubscribe()** grid method. Then the shared variable records are pushed to the consumer/subscriber components. Subsequently, the component resource will invoke **terminate()** method of all the four active threads. After that, the component resource set a termination time for itself so that the GT4 container will destroy the component resource. The component resource will then notify the organizer service the termination status of itself. If the simulation description file has specified simulation results to be collected, such simulation results will be sent back to the organizer. The organizer will wait until all the component resources are terminated and save the simulation results at a specified location. Finally, the organizer resource set termination time as an immediate termination so that the GT4 container will destroy the organizer resource.

Figure 6.4 Sequence diagram for the premature simulation termination

Figure 6.5 Sequence diagram of the normal simulation termination

6.4 Summary

This chapter details the simulation cycle in SOAr-DSGrid. The organizer transforms a simulation description into a simulation execution. The simulation execution is carried out distributively, and the communication between component services is on a peer-to-peer basis without any intermediary parties. The component services coordinate among themselves through the common component service interfaces and runtime elements in the primitive component resource. The simulation execution could either be prematurely terminated upon the client's instruction or be terminated when a termination condition is satisfied.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

In-depth study and survey on Grid technologies, Web/Grid Services specifications, workflow languages and execution models, component technologies and architectures, simulation architectures/frameworks, and XML related technologies have been conducted. The literature survey discovered that existing frameworks or models do not provide all elements for distributed simulation using component approach on the Grid as explained in detail in Chapter 2. Either there is no built-in runtime support for simulation, or the component orchestration and composition is not properly addressed. Thus, it is necessary to develop a new framework for component-based distributed simulation on the Grid. In order to relieve the component developers and simulation application developers from Java web services details, different views for developments and executions of components and simulation applications are necessary.

Features

The proposed framework, SOAr-DSGrid has the following features:

- It provides component-based view for component development and simulation application development, while service-oriented view for simulation execution;
- The components are developed by providing component interfaces conforming to a specific XML Schema and developing operations supported by the component;

- The components are generated, built, and deployed by SOAr-DSGrid service generator, and registered with the index service;
- Simulation applications are built by orchestrating existing components. No modification on the existing components is required when orchestrating them into a simulation application;
- The execution of a simulation is initialized and instantiated by the organizer service. The simulation execution is self-coordinated among components;
- The framework supports shared variable management. A component could share variables with other components and/or subscribe variables of other components;
- The composition of components in this framework is different from workflow as the composition is dynamic by introducing the rules and local rule engines. Operation rules and routing rules are actively involved to enable dynamic composition at run time instead of design time. These rules make components possess certain degree of intelligence in determining next operation and next component;
- The data flow is decentralized at run time. The data transfer is in peer-to-peer manner as source components have knowledge of target components at run time;
- The framework runtime provides built-in time management for simulation components; and
- Simulation components can be distributed Internet-wide with GT4 and the framework runtime as common middlewares.

Achievements

In order to realize the abovementioned features, the following work has been done during the master study:

- Conceived the architecture of the framework. This includes the definition of five roles of the framework, i.e. the client, the component provider, the organizer, the index service, and the repository;

- Defined the simulation worldview in the framework. The communication in the framework is in process-oriented simulation worldview;
- Standardized formats for component interfaces, composite component implementation files, simulation description files, and entity type files. All these formats are in XML schema;
- Designed and implemented the structure of a primitive component resource. A primitive component resource comprises both component-general elements and component-specific elements. All the three major parts of a component service, i.e., the resource, the resource home, and the service implementation, are implemented;
- Designed and implemented the service generator for the component provider to transform the component interface files in component-based view into component services in the service-oriented view. The service generator can be used to generate, build, and deploy a component service. The relevant GUI is also in place for the component provider;
- Designed and implemented the organizer for the client to transform the simulation description files in component-based view into a simulation execution in the service-oriented view. The organizer parses simulation description files, creates simulation session space, instantiates component resource instances, triggers simulation execution of component resources, and collects simulation results. The relevant GUI is also provided for the client;
- Specified procedures involved in the framework. These include model design procedures, simulation initialization procedures, simulation execution procedures, and simulation termination procedures;
- Designed and implemented the index service. The index service is capable of handling directory services for component service, component resource in-

stances, and four kinds of files in the repository. Basic directory services include registration, deregistration, query, and update; and

- Designed and implemented a notification mechanism for simulation termination.

Usability

With the abovementioned work on SOAr-DSGrid framework, SOAr-DSGrid is capable of facilitating the simulation component development and distributed simulation execution on the Grid. It has been used to experiment the shared variable management as elaborated in section 5.3 and Appendix B.1 to B.5. It can also be used for wafer manufacturing simulation as shown in section 4.3 and Appendix D.1 to D.3.

7.2 Outlook of Future Work

The master study completes the implementation of the framework with primitive components and basic roles. The future work includes the implementation of composite component, the time management in composite component execution, and the shared variable management in composite component. The composite component faces more complex issues than primitive component. The determination of the simulation time of a composite component will not be as trivial as that of a primitive component because the constituent components of a composite component may be at different simulation time. The time management of a composite component will be closely coupled with the shared variable management of a composite component.

In the shared variable management, an adaptive algorithm, which allows the reader and/or the owner to switch between Pull with FutureList and Push dynamically, could be developed to improve the performance.

Currently SOAr-DSGrid uses BeanShell and rule configuration to implement a simple rule engine. A more sophisticated rule engine such as Drools [55] can be incorporated into SOAr-DSGrid to support more complex rules.

APPENDICES

APPENDIX A : SCHEMAS

A.1 Entity Type Schema

```
<?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.fcbsdsg.org/entity" targetNamespace="http://www.fcbsdsg.org/entity" element-
  FormDefault="unqualified" attributeFormDefault="unqualified">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Entity Type Schema
      Copyright 2004-2008 PDCC NTU Singapore.
    </xsd:documentation>
  </xsd:annotation>
  <!-- The root element -->
  <xsd:element name="entity" type="ent:entityType"/>
  <!-- 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="property" 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: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:schema>
```

A.2 Component Interface Schema

```
<?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://soardsgrid/component/primitive/interface" targetName-
  space="http://soardsgrid/component/primitive/interface" elementFormDefault="unqualified" attribute-
  FormDefault="unqualified">
```

```

<xsd:annotation>
  <xsd:documentation>
    Component Interface Schema
    Copyright 2004-2008 PDCC NTU Singapore.
  </xsd:documentation>
</xsd:annotation>
<!-- root element -->
<xsd:element name="component">
  <xsd:complexType>
    <xsd:complexContent>
      <xsd:extension base="comp:componentType"/>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:element>
<!-- 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>
    <xsd:element name="description" type="xsd:string"/>
    <xsd:element name="operations" maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="name" type="xsd:string"/>
          <xsd:element name="description" type="xsd:string"/>
          <xsd:element name="inputEntity" minOccurs="0">
            <xsd:complexType>
              <xsd:sequence>
                <xsd:element name="type" type="xsd:string" maxOc-
curs="unbounded"/>
              </xsd:sequence>
            </xsd:complexType>
          </xsd:element>
          <xsd:element name="outputEntity" minOccurs="0">
            <xsd:complexType>
              <xsd:sequence>
                <xsd:element name="type" type="xsd:string" maxOc-
curs="unbounded"/>
              </xsd:sequence>
            </xsd:complexType>
          </xsd:element>
          <!-- operation description -->
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="variables" type="comp:attributeType" maxOccurs="unbounded"/>
    <!-- Component description -->
  </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:attribute name="category" use="required">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="generator"/>
        <xsd:enumeration value="processor"/>
        <xsd:enumeration value="collector"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:attribute>

```

```

    </xsd:attribute>
  </xsd:complexType>
  <xsd:complexType name="attributeType">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="type" type="xsd:string"/>
      <xsd:element name="description" type="xsd:string"/>
      <xsd:element name="defaultValue" type="xsd:string" minOccurs="0"/>
      <xsd:element name="mode">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:enumeration value="local"/>
            <xsd:enumeration value="localShared"/>
            <xsd:enumeration value="remoteShared"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
      <!-- Description of variable -->
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>

```

A.3 Composite Component Implementation Schema

```

<?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">
      Composite Component Implementation Schema
      Copyright 2004-2008 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" type="impl:componentInstanceType" maxOc-
curs="unbounded"/>
      <xsd:element name="composition">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="entityFlows" type="impl:entityFlowType" minOccurs="0"
maxOccurs="unbounded"/>
            <xsd:element name="variableFlows" type="impl:variableFlowType" nilla-
ble="true" minOccurs="0" maxOccurs="unbounded"/>
          </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="setupVariables" minOccurs="0" 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="operationRules" type="impl:ruleConfiguration" minOccurs="0" maxOc-
curs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="componentType" type="xsd:string" use="required"/>
    <xsd:attribute name="version" type="xsd:string" use="required"/>
  </xsd:complexType>
  <xsd:complexType name="entityFlowType">
    <xsd:sequence>
      <xsd:element name="entryComponent" type="xsd:string"/>
      <xsd:element name="entityLinks" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="source" type="xsd:string"/>
            <xsd:element name="targets" type="impl:Target" maxOccurs="unbounded"/>
            <xsd:element name="routingRule" type="impl:ruleConfiguration" minOc-
curs="0"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="entityType" type="xsd:string" use="required"/>
  </xsd:complexType>
  <xsd:complexType name="Target">
    <xsd:sequence>
      <xsd:element name="target" type="xsd:string"/>
      <xsd:element name="travelTime" type="xsd:double"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="variableFlowType">
    <xsd:sequence>
      <xsd:element name="provider" type="impl:Provider"/>
      <xsd:element name="subscribers" type="impl:Subscriber" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="Provider">
    <xsd:sequence>
      <xsd:element name="providerName" type="xsd:string"/>
      <xsd:element name="variableName" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="Subscriber">
    <xsd:sequence>
      <xsd:element name="subscriberName" type="xsd:string"/>
      <xsd:element name="variableName" type="xsd:string"/>
    </xsd:sequence>
  </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="configVars" type="impl:configurationVariableType" maxOc-
curs="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>

```

A.4 Simulation Description Schema

```

<?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:sim="http://soardsgrid/organizer/sim" targetNamespace="http://soardsgrid/organizer/sim" element-
FormDefault="unqualified" attributeFormDefault="unqualified">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            Simulation Description Schema
            Copyright 2004-2008 PDCC NTU Singapore.
        </xsd:documentation>
    </xsd:annotation>
    <!-- The root element -->
    <xsd:element name="simulation" type="sim:Simulation"/>
    <xsd:simpleType name="Version">
        <xsd:restriction base="xsd:string">
            <xsd:pattern value="[0-9]+\.[0-9]+"/>
        </xsd:restriction>
    </xsd:simpleType>
    <xsd:complexType name="Simulation">
        <xsd:sequence>
            <!-- components involved in the simulation from the view point of the client -->
            <xsd:element name="componentInstances" type="sim:ComponentInstance" maxOc-
curs="unbounded"/>
            <xsd:element name="composition" type="sim:Composition"/>
            <!-- simulation results to be collected from components -->
            <xsd:element name="results" type="sim:Result" maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="sessionName" type="xsd:string" use="required"/>
    </xsd:complexType>
    <xsd:complexType name="Result">
        <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:complexType>
<xsd:complexType name="ComponentInstance">
  <xsd:sequence>
    <xsd:element name="setupVariables" type="sim:SetupVariable" minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="operationRules" type="sim:RuleConfiguration" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string" use="required"/>
  <xsd:attribute name="componentType" type="xsd:string" use="required"/>
  <xsd:attribute name="version" type="sim:Version" use="required"/>
</xsd:complexType>
<xsd:complexType name="SetupVariable">
  <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:complexType name="Composition">
  <xsd:sequence>
    <xsd:element name="entityFlows" type="sim:EntityFlow" nillable="true" minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="variableFlows" type="sim:VariableFlow" nillable="true" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="EntityFlow">
  <xsd:sequence>
    <xsd:element name="entityLinks" type="sim:EntityLink" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="entityType" type="xsd:string" use="required"/>
</xsd:complexType>
<xsd:complexType name="EntityLink">
  <xsd:sequence>
    <xsd:element name="source" type="xsd:string"/>
    <xsd:element name="targets" type="sim:Target" maxOccurs="unbounded"/>
    <xsd:element name="routingRule" type="sim:RuleConfiguration" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="Target">
  <xsd:sequence>
    <xsd:element name="target" type="xsd:string"/>
    <xsd:element name="travelTime" type="xsd:double"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="VariableFlow">
  <xsd:sequence>
    <xsd:element name="provider" type="sim:Provider"/>
    <xsd:element name="subscribers" type="sim:Subscriber" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="Provider">
  <xsd:sequence>
    <xsd:element name="providerName" type="xsd:string"/>
    <xsd:element name="variableName" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="Subscriber">
  <xsd:sequence>
    <xsd:element name="subscriberName" type="xsd:string"/>
    <xsd:element name="variableName" type="xsd:string"/>
  </xsd:sequence>

```

```

</xsd:complexType>
<xsd:complexType name="RuleConfiguration">
  <xsd:sequence>
    <xsd:element name="configVars" type="sim:ConfigurationVariable" maxOc-
curs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string" use="required"/>
  <xsd:attribute name="type" type="xsd:string" use="required"/>
  <xsd:attribute name="url" type="xsd:string" use="optional"/>
  <xsd:attribute name="applyToEntity" type="xsd:string" use="required"/>
</xsd:complexType>
<xsd:complexType name="ConfigurationVariable">
  <xsd:choice>
    <xsd:sequence>
      <xsd:element name="type" type="xsd:string" nillable="true"/>
      <xsd:element name="value" type="xsd:string" nillable="true"/>
    </xsd:sequence>
    <xsd:sequence>
      <xsd:element name="objectName" type="xsd:string" nillable="true"/>
      <xsd:element name="attributeName" type="xsd:string" nillable="true"/>
    </xsd:sequence>
  </xsd:choice>
  <xsd:attribute name="name" type="xsd:string" use="required"/>
  <xsd:attribute name="mappingType" type="sim:MappingType" use="required"/>
</xsd:complexType>
<xsd:simpleType name="MappingType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="mapToValue"/>
    <xsd:enumeration value="mapToVariable"/>
    <xsd:enumeration value="mapToAttribute"/>
  </xsd:restriction>
</xsd:simpleType>
</xsd:schema>

```

A.5 ServiceGen Schema

```

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:gen="http://soardsgrid/generator/servicegen" targetName-
space="http://soardsgrid/generator/servicegen" elementFormDefault="unqualified" attributeFormDe-
fault="unqualified">

```

```

  <xs:annotation>
    <xs:documentation>

```

The elements specified in this file are used for service generation. This information are independent of component interface file. The component interface file describes the component information. This file contains information used to transform component into web service.

```

    </xs:documentation>
  </xs:annotation>
  <xs:element name="serviceGen" type="gen:serviceGenType"/>
  <xs:complexType name="serviceGenType">
    <xsd:sequence>
      <!--

```

serviceName: This is the name assigned to the service. It could be a simple identifier or a path.

The name can be ab/cd/ef/ServiceA. There should be no pending or trailing "/" in service name.

This name will be used in deploy-server.wsdd file to specify the name of the service. It will be part of the service URL.

http://127.0.0.1/wsrf/services/ab/cd/ef/ServiceA.

The serviceName will be used to generate deploy-server.wsdd and deploy-jndi-config.xml.
 If not specified, componentName from component interface file
 (/comp:component/@type) will be used as serviceName.
 This variable goes into intermediate XML file.

```
-->
<xs:element name="serviceName" type="xs:string" minOccurs="0"/>
<!--
```

componentNS: the namespace assigned to the component.
 This value will be set to the namespace of the component in the generated WSDL.
 It will also be used to generate the namespace2package.mappings file if there is no explicit
 namespace to package mappings in this serviceGen.xml.
 This variable goes into intermediate XML file.

```
-->
<xs:element name="componentNS" type="xs:string"/>
<!--
```

servicePackage is of format ab.cd.ef, specifying the service package.
 packageDir will be of format ab/cd/ef.
 packageDir: This is the place to stored the generated ServiceImpl java file and Resource
 java file.

This directory must have two subdirectory:
 1) impl: The directory contains the generated service implementation, resource class,
 and resource home.
 2) etc: This directory may contain other xml files, and can be empty.
 The packageDir itself will contain the generated deploy-server.wsdd and deploy-jndi-
 config.xml.

This variable goes into intermediate XML file.

```
-->
<xs:element name="servicePackage" type="xs:string"/>
<!--
```

serviceImplPackage: The preferred package name. Notice that if the package does
 not end with '.impl', SOArDSGrid service generator will append the '.impl'
 to the package impl package name. The package directory should contain deploy-server-
 config.wsdd and deploy-jndi-config.xml.
 The package directory should also contain an etc subdirectory besides impl subdirectory.
 If not specified, servicePackage + ".impl" will be used.
 If specified, it must be servicePackage + ".impl".
 This variable goes into intermediate XML file.

```
-->
<xs:element name="serviceImplPackage" type="xs:string" minOccurs="0"/>
<!--
```

componentClassName: This is the component class name, the service implementation class
 name
 without .class extension.
 This variable goes into intermediate XML file.

```
-->
<xs:element name="componentClassName" type="xs:string"/>
<!--
```

resourceClassName: This is the resource class name without .class extension.
 If not specified, it will be {componentClassName}Resource.
 This variable goes into intermediate XML file.

```
-->
<xs:element name="resourceClassName" type="xs:string" minOccurs="0"/>
<!--
```

resourceHomeClassName: This is the resource home class name without .class extension.
 If not specified, it will be {resourceClassName}Home.
 This variable goes into intermediate XML file.

```
-->
<xs:element name="resourceHomeClassName" type="xs:string" minOccurs="0"/>
<!--
```

resourcePropNS is the resource properties namespace.
 If not specified, the componentNS will be used by default.
 This variable goes into intermediate XML file.

```
-->
```

```

<xs:element name="resourcePropNS" type="xs:string" minOccurs="0"/>
<!--
  This is the base directory for generated source file.
  The base directory will contain the following subdirectory:
    1) All the generated files will be generated in current directory.
    2) classes: All the compiled classes will be in this directory.
    3) build.xml: This will be a predefined build file which is generic for every service.
  If not specified, the default value $SOARDSGRID_REPO/service will be used.
  This variable goes into intermediate XML file.
-->
<xs:element name="sourceBase" type="xs:string" minOccurs="0"/>
<!--
  schemaRoot specifies the root path of the generated schema.
  If not specified, the default will be $SOARDSGRID_REPO/share/schema.
  This variable does not go into intermediate XML file.
-->
<xs:element name="schemaRoot" type="xs:string" minOccurs="0"/>
<!--
  schemaPath is the relative path with base directory {schemaRoot}.
  If not specified, the default will be the {packageDir}.
  This variable does not go into intermediate XML file.
-->
<xs:element name="schemaPath" type="xs:string" minOccurs="0"/>
<!--
  garId will be used as gar file name. The gar file name will be {garId}.gar.
  The garId will also be used to undeploy the service and deploy the service.
  In $GLOBUS_LOCATION/etc, there should be one directory named by {garId}.
  In $GLOBUS_LOCATION/lib, there should be {garId}.jar and {garId}_stubs.jar
  In $GLOBUS_LOCATION/share/schema, the schema corresponding to the {garId} is
there.
  If not specified, the default will be servicePackage with format ab_cd_ef.
  This variable does not go into intermediate XML file.
-->
<xs:element name="garId" type="xs:string" minOccurs="0"/>
<!--
  This is to specify which kind of shared variable scheme is preferred. The default is primitivepull.
  This determines the base Service implementation class and base Resource class.
  If not specified, the default primitivepull will be used.
  This variable goes into intermediate XML file.
-->
<xs:element name="sharedVariableScheme" minOccurs="0">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="primitivepull"/>
      <xs:enumeration value="primitivepush"/>
      <xs:enumeration value="singlepull"/>
      <xs:enumeration value="adaptivepullpush"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
<!--
  interfaceName will be the name of the generated WSDL file, the {interface-
Name}PortType will be the portType name in the generated WSDL.
  The {interfaceName}_flattened.wsdl, {interfaceName}_bindings.wsdl, and {interface-
Name}_service.wsdl will be generated.
  If not specified, the componentClassName will be used as interfaceName.
  This variable does not go to intermediate XML file.
-->
<xs:element name="interfaceName" type="xs:string" minOccurs="0"/>
<!--
  This is to specify the namespace to package mapping. It will be used to generate pack-
age2namespace.mappings file.

```

This mapping will be used in stub generation.

```
-->
<xs:element name="namespaces2packages" minOccurs="0">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="ns2pkg" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="ns" type="xs:string"/>
            <xs:element name="pkg" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:schema>
```

A.6 Intermediate XML File Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSpy v2005 rel. 3 U (http://www.altova.com) by XINJUN (EMBRACE) -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:intm="http://soardsgrid/component/primitive/intermediate" targetName-
  space="http://soardsgrid/component/primitive/intermediate" elementFormDefault="qualified" attribute-
  FormDefault="unqualified">
  <xs:element name="component" type="intm:componentType">
    <xs:annotation>
      <xs:documentation>The schema for the intermediate XML file</xs:documentation>
    </xs:annotation>
  </xs:element>
  <xs:complexType name="componentType">
    <xs:sequence>
      <xs:element name="serviceName" type="xs:string"/>
      <xs:element name="componetNS" type="xs:string"/>
      <xs:element name="servicePackage" type="xs:string"/>
      <xs:element name="serviceImplPackage" type="xs:string"/>
      <xs:element name="componentClassName" type="xs:string"/>
      <xs:element name="resourceClassName" type="xs:string"/>
      <xs:element name="resourceHomeClassName" type="xs:string"/>
      <xs:element name="resourcePropNS" type="xs:string"/>
      <xs:element name="sourceBase" type="xs:string"/>
      <xs:element name="wsrfSchemaRoot" type="xs:string"/>
      <xs:element name="soardsgridSchemaRoot" type="xs:string"/>
      <xs:element name="sharedVariableScheme" type="xs:string"/>
      <xs:element name="defaultSchemaPrefix" type="xs:string"/>
      <xs:element name="variables">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="variable" type="intm:variableType" maxOc-
curs="unbounded"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="variableType">
    <xs:sequence>
```



```

<xs:element name="name" type="xs:string"/>
<xs:element name="description" type="xs:string" minOccurs="0"/>
<xs:element name="javaType" type="xs:string"/>
<xs:element name="xsdType" type="xs:string"/>
<xs:element name="name2" type="xs:string"/>
<xs:element name="name3" type="xs:string"/>
</xs:sequence>
</xs:complexType>

</xs:schema>

```

APPENDIX B : SAMPLE FILES

B.1 OperationA (Owner's Operation)

```

/**
 * This is a sample owner.
 */
package soardsgrid.operation.test1;

import java.awt.BorderLayout;
import java.awt.Container;
import java.lang.management.ManagementFactory;
import java.lang.management.ThreadMXBean;
import java.util.Iterator;
import java.util.List;
import java.util.Set;
import javax.swing.BorderFactory;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import soardsgrid.commandline.CommandLineTypes;
import soardsgrid.component.primitivepull.Operation;
import soardsgrid.component.primitivepull.OperationCompleteEvent;
import soardsgrid.component.primitivepull.SharedVariable;
import soardsgrid.component.primitivepull.service.impl.PrimitiveComponentResource;
import soardsgrid.util.Configuration;
/**
 * @author chen0081
 */
public class CopyOfOperationA extends Operation {
    private static Log log = LogFactory.getLog(CopyOfOperationA.class);
    private Configuration config = Configuration.getInstance();
    private long processingTime;
    private double[] updatePattern;
    private double simTime;
    private double lookahead;
    private ThreadMXBean timeBean;
    private void init() {
        timeBean = ManagementFactory.getThreadMXBean();
        if (!timeBean.isCurrentThreadCpuTimeSupported()) {
            log.warn("CPU Usage monitoring is not available!");
        } else {
            timeBean.setThreadCpuTimeEnabled(true);
        }
        log.info("init successful in OperationA.");
    }

    public OperationA() {

```



```

        super();
        init();
    }

    public OperationA(PrimitiveComponentResource resource) {
        super(resource);
        init();
        options = resource.getOptions();
        if(options==null || !options.containsKey(CommandLineTypes.OWNER_PROCESSING_TIME)) {
            processingTime = config.getOwnerProcessingTime();
        } else {
            processingTime
Long.parseLong((String)options.get(CommandLineTypes.OWNER_PROCESSING_TIME));
        }
        //processingTime *= 1000000;

        if(options== null || !options.containsKey(CommandLineTypes.UPDATE_PATTERN)) {
            updatePattern = config.getUpdatePattern();
        } else {
            String pattern = (String) options.get(CommandLineTypes.UPDATE_PATTERN);
            String[] patterns = pattern.split(",");
            if(patterns != null && patterns.length > 0) {
                int size = patterns.length;
                updatePattern = new double[size];
                for(int i=0; i<size; i++) {
                    updatePattern[i] = Integer.parseInt(patterns[i]);
                }
            }
        }

        if(options==null || !options.containsKey(CommandLineTypes.SIMULATION_END_TIME)) {
            simTime = config.getSimulationEndTime();
        } else {
            String endTime = (String) options.get(CommandLineTypes.SIMULATION_END_TIME);
            simTime = Double.parseDouble(endTime);
        }

        if(options== null || !options.containsKey(CommandLineTypes.LOOKAHEAD)) {
            if(options==null) {
                log.debug("Null options. Trying to get lookahead from configuration file.");
            } else {
                log.debug("options does not contain lookahead. Trying to get lookahead from configuration file.");
            }
            lookahead = config.getReaderLookahead();
        } else {
            String la = (String) options.get(CommandLineTypes.LOOKAHEAD);
            lookahead = Double.parseDouble(la);
        }
    }

    private long processing(long count) {
        /**
         * Spin-loop emulates the processing required.
         */
        long startCpu = timeBean.getCurrentThreadUserTime();
        int k = 0;
        for(long i = 0; i < count; i++) {
            //Math.sin(i);
            k = (int)(i+1) % 1981 * 1983;
        }
        long endCpu = timeBean.getCurrentThreadUserTime();
        if(k == 0) {}
        return (endCpu - startCpu);
    }

    public void execute() {
        long loopCount = processingTime * 100000;
        //startOperation();
        if(log.isDebugEnabled()) {
            StringBuffer sb = new StringBuffer();

```

```

        sb.append("Owner enter execute(). ownerProcessingTime=").append(processingTime/1000000)
        .append(". simEndTime=").append(simTime).append(" Lookahead      from
reader=").append(lookahead).append("\nUpdate Pattern: ");
        for(int i=0; i<updatePattern.length; i++) {
            sb.append(updatePattern[i]).append(",");
        }
        log.debug(sb.toString());
    }
    //String reader = "svexp:reader1";
    double time = 0;
    double[] updateIntervals = updatePattern;
    int patternSize = updateIntervals.length;
    /**
     * Prepare for the Swing JTextArea
     */
    JTextArea ta = new JTextArea();
    ta.setText("Starting simulation: ");
    ta.setBorder(BorderFactory.createLoweredBevelBorder());
    ta.setEnabled(true);
    ta.setEditable(false);
    ta.setColumns(30);
    ta.setLineWrap(true);
    ta.setRows(5);
    ta.setWrapStyleWord(true);
    ta.setAutoscrolls(true);

    String ownerKeyStringColon = ownerComponent.getKeyStringColon();
    JFrame win = new JFrame(ownerKeyStringColon);
    win.setLocation(100, 100);
    win.setAlwaysOnTop(true);
    Container contentPane = win.getContentPane();
    win.setResizable(false);
    contentPane.add(new JScrollPane(ta), BorderLayout.CENTER);
    win.pack();
    win.setVisible(true);
    ta.setCaretPosition(ta.getDocument().getLength());

    String svName = "var3";
    int oldValue;
    int newValue;
    double oldTime;
    double newTime;
    long processing = 0;
    int loop = 0;
    long start = System.nanoTime();

    while(true) {
        for(int j=0; j<patternSize; j++) {
            if(time >= simTime) {break;}
            loop++;
            log.debug("Owner is going to update time to " + (time + updateIntervals[j]));
            try {
                oldValue = ((Integer)ownerComponent.getVariable(svName)).intValue();
                newValue = oldValue + 1;
                ownerComponent.setVariable(svName, Integer.valueOf(newValue));
                long cpuTime = processing(loopCount);
                processing += cpuTime;
                log.debug("CPU Time=" + cpuTime);
                oldTime = time;
                newTime = time + updateIntervals[j];
                time = newTime;
                log.debug("Start to update and synchronize variable var3. Update time from " + oldTime + " to "
+ newTime);
                ownerComponent.setSimulationTime(time);
                ownerComponent.synchronizeSharedVariables();
                ta.append("\nnewTime=" + newTime + ", newValue=" + newValue);
                ta.setCaretPosition(ta.getDocument().getLength());
            } catch (Exception e) {
                log.error(e, e);
                e.printStackTrace();
            }
        }
    }

```

```

    }
    }
    if(time >= simTime) {break;}
}

long end = System.nanoTime();
ta.append("\n" + ownerKeyStringColon + " completes execution.");
ta.append("\n" + ownerKeyStringColon + " execution time=" + (end - start)/1000000 + " milliseconds. Average unit processingTime=" + processing/loop/1000000);
ta.setCaretPosition(ta.getDocument().getLength());
log.info("OperationA (Owner) execution time=" + (end - start)/1000000 + " milliseconds. Average unit processingTime=" + processing/loop/1000000);

SharedVariable v3 = (SharedVariable) ownerComponent.getSharedVariables().get(svName);
if(v3==null) {
    log.error("Cannot find SharedVariable " + svName);
    return;
}

int subscriberIndex = 0;
StringBuffer logBuf = new StringBuffer();
logBuf.append("numOfDirectReturns=").append(v3.getNumsOfDirectReturns().get(subscriberIndex))
    .append("; NumOfRequests=").append(v3.getNumsOfRequests().get(subscriberIndex))
    .append("; NumOfSolicitedPushes=").append(v3.getNumsOfSolicitedPushes().get(subscriberIndex))
    .append("; NumOfUnsolicitedPushes=").append(v3.getNumsOfUnsolicitedPushes().get(subscriberIndex))
    .append("; NumOfSwitches=").append(v3.getNumsOfSwitches().get(subscriberIndex))
    .append("; NumOfUpdates=").append(v3.getNumOfUpdates())
    .append("; NumOfFossilCollect=").append(v3.getNumOfFossilCollections()).append("\n");

List requests = v3.getRequestTimes();
int numOfRequests = requests.size();
logBuf.append("NumOfRequests received by owner: " + numOfRequests);
for(int i=1; i<numOfRequests; i++) {
    if(i % 5 == 0) {logBuf.append("\n");}
    Double last = (Double)requests.get(i-1);
    Double current = (Double)requests.get(i);
    logBuf.append("          reqInterval").append(i).append(")=").append(current.doubleValue()
last.doubleValue());
}
log.info(logBuf.toString());
ownerComponent.terminate(true);
}

public void preExecute () {
    // acquire execution resource required for this execution.
    ownerComponent.reserveOperationResource(1);
}

public void postExecute(OperationCompleteEvent event) {
    ownerComponent.releaseOperationResource(1);
}
}

```

B.2 CompA Component Interface

```

<?xml version="1.0" encoding="UTF-8"?>
<comp:component xmlns:comp="http://soardsgrid/component/primitive/interface"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://soardsgrid/component/primitive/interface
D:\soardsgridRepo\share\schema\soardsgrid\component\ComponentInterface.xsd" provider="Xinjun" category="processor" type="ReaderComp" version="1.0" isComposite="false">
    <description>This is a component interface for component CompA</description>
    <operations>
        <name>soardsgrid.operation.OperationA</name>

```

```

    <description>This is a sample operation of CompA</description>
  </operations>
  <operations>
    <name>soardsgrid.operation.OperationX</name>
    <description>This is another dummy sample operation of CompA</description>
  </operations>
  <variables>
    <name>var1</name>
    <type>java.lang.String</type>
    <description>This is a String variable</description>
    <mode>local</mode>
  </variables>
  <variables>
    <name>var2</name>
    <type>int</type>
    <description>This is an int variable</description>
    <mode>local</mode>
  </variables>
  <variables>
    <name>var3</name>
    <type>int</type>
    <description>This is a shared variable</description>
    <mode>localShared</mode>
  </variables>
</comp:component>

```

B.3 Shared Variable Simulation Description File

```

<?xml version="1.0" encoding="UTF-8"?>
<simulation xmlns="http://soardsgrid/organizer/sim" xmlns:sim="http://soardsgrid/organizer/sim"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" sessionName="svexp">
  <componentInstances name="reader1" componentType="ReaderComp" version="0.1">
    <setupVariables>
      <name>v2</name>
      <type>java.lang.String</type>
      <value>Reader1 V2Value</value>
    </setupVariables>
  </componentInstances>
  <componentInstances name="owner1" componentType="OwnerComp" version="0.1">
    <setupVariables>
      <name>var3</name>
      <type>int</type>
      <value>1</value>
    </setupVariables>
  </componentInstances>
  <composition>
    <variableFlows>
      <provider>
        <providerName>owner1</providerName>
        <variableName>var3</variableName>
      </provider>
      <subscribers>
        <subscriberName>reader1</subscriberName>
        <variableName>v3</variableName>
      </subscribers>
    </variableFlows>
  </composition>
</simulation>

```

B.4 CompA ServiceGen

```
<?xml version="1.0" encoding="UTF-8"?>
<gen:serviceGen xmlns:gen="http://soardsgrid/generator/servicegen"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://soardsgrid/generator/servicegen
D:\soardsgridRepo\share\schema\soardsgrid\generator\ServiceGen.xsd">
  <serviceName>ReaderComp</serviceName>
  <componentNS>http://proto1.component.soardsgrid/CompA</componentNS>
  <servicePackage>soardsgrid.component.proto1.compA.service</servicePackage>
  <componentClassName>CompA</componentClassName>
  <sourceBase>D:\soardsgridRepo\temp</sourceBase>
  <namespaces2packages>
    <ns2pkg>
      <ns>http://proto1.component.soardsgrid/CompA</ns>
      <pkg>soardsgrid.component.proto1.stubs.compA</pkg>
    </ns2pkg>
    <ns2pkg>
      <ns>http://proto1.component.soardsgrid/CompA/bindings</ns>
      <pkg>soardsgrid.component.proto1.stubs.compA.bindings</pkg>
    </ns2pkg>
    <ns2pkg>
      <ns>http://proto1.component.soardsgrid/CompA/service</ns>
      <pkg>soardsgrid.component.proto1.stubs.compA.service</pkg>
    </ns2pkg>
  </namespaces2packages>
</gen:serviceGen>
```

B.5 Sample Intermediate XML File

```
<component>
  <serviceName>ReaderComp</serviceName>
  <componentNS>http://proto1.component.soardsgrid/CompA</componentNS>
  <servicePackage>soardsgrid.component.proto1.compA.service</servicePackage>
  <serviceImplPackage>soardsgrid.component.proto1.compA.service.impl</serviceImplPackage>
  <componentClassName>CompA</componentClassName>
  <resourceClassName>CompAResource</resourceClassName>
  <resourceHomeClassName>CompAResourceHome</resourceHomeClassName>
  <resourcePropNS>http://proto1.component.soardsgrid/CompA</resourcePropNS>
  <sourceBase>D:\soardsgridRepo\temp</sourceBase>
  <wsrfSchemaRoot>../../../../..</wsrfSchemaRoot>
  <soardsgridSchemaRoot>../../../../..</soardsgridSchemaRoot>
  <sharedVariableScheme>primitivepull</sharedVariableScheme>
  <defaultSchemaPrefix>xs</defaultSchemaPrefix>
  <variables>
    <variable>
      <name>var1</name>
      <description>This is a String variable</description>
      <javaType>java.lang.String</javaType>
      <xsdType>xs:string</xsdType>
      <name2>Var1</name2>
      <name3>VAR1</name3>
    </variable>
    <variable>
      <name>var2</name>
      <description>This is an int variable</description>
      <javaType>int</javaType>
      <xsdType>xs:int</xsdType>
```

```

        <name2>Var2</name2>
        <name3>VAR2</name3>
    </variable>
    <variable>
        <name>var3</name>
        <description>This is another int variable</description>
        <javaType>int</javaType>
        <xsdType>xs:int</xsdType>
        <name2>Var3</name2>
        <name3>VAR3</name3>
    </variable>
</variables>
</component>

```

APPENDIX C : TEMPLATES

C.1 Resource Template

```

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" >
<xsl:output method="text" indent="yes" encoding="iso-8859-1"/>
<xsl:template match="/">
/**
 * NTU PDCC SOArDSGrid
 * This class is automatically generated by SOArDSGrid service generator.
 */
package <xsl:value-of select="component/serviceImplPackage"/>;

import javax.xml.namespace.QName;

import org.globus.wsrf.Resource;
import org.globus.wsrf.ResourceIdentifier;
import org.globus.wsrf.ResourceLifetime;
import org.globus.wsrf.ResourceProperties;
import org.globus.wsrf.ResourceProperty;
import org.globus.wsrf.impl.ReflectionResourceProperty;
import org.globus.wsrf.impl.SimpleResourcePropertyMetaData;
import org.globus.wsrf.impl.SimpleResourcePropertySet;

/**
 * @author chen0081
 */
<xsl:variable name="resourceClassName" select="component/resourceClassName"/>
public class <xsl:variable name="resourceClassName"/> extends soards-
grid.component.primitivepull.service.impl.PrimitiveComponentResource implements Resource, Resource-
Lifetime, ResourceIdentifier, ResourceProperties, java.io.Serializable {

    private static final long serialVersionUID = -1418789403151782808L;
    public final QName RP_SET = new QName(propNS, "<xsl:variable name='resourceClassName'/>");
    String propNS = "<xsl:value-of select='component/resourcePropNS'/>";

    <xsl:for-each select="component/variables/variable">
    // <xsl:value-of select="description"/>
    protected <xsl:value-of select="type"/> <xsl:value-of select="name"/>;

```

```

QName PR_<xsl:value-of select="name3"/> = new QName(propNS, <xsl:value-of select="name3"/>);

</xsl:for-each>
public <xsl:variable name="resourceClassName"/>() {
    super();
    resourceInstance = this;
}

protected void initialize(Object resourceKey) {
    this.key = resourceKey;
    this.propSet = new SimpleResourcePropertySet(RP_SET);
    ResourceProperty prop = null;

    try {
        prop = new ReflectionResourceProperty(
            SimpleResourcePropertyMetaData.TERMINATION_TIME, this);
        this.propSet.add(prop);
        prop = new ReflectionResourceProperty(
            SimpleResourcePropertyMetaData.CURRENT_TIME, this);
        this.propSet.add(prop);
        <xsl:for-each select="component/variables/variable">
            prop = new ReflectionResourceProperty(PR_<xsl:value-of select="name3"/>, this);
            this.propSet.add(prop);
        </xsl:for-each>

    } catch (Exception e) {
        throw new RuntimeException(e.getMessage());
    }
}

<xsl:for-each select="component/variables/variable">
/**
 * @return Returns the <xsl:value-of select="name"/>
 */
public <xsl:value-of select="type"/> get<xsl:value-of select="name2"/>() {
    return <xsl:value-of select="name"/>;
}

/**
 * @param <xsl:value-of select="name"/> The <xsl:value-of select="name"/> to set.
 */
public void set<xsl:value-of select="name2"/>(<xsl:value-of select="type"/> <xsl:value-of se-
lect="name"/>) {
    this.<xsl:value-of select="name"/> = <xsl:value-of select="name"/>;
}
</xsl:for-each>
</xsl:template>
</xsl:stylesheet>

```

C.2 ResourceHome Template

```

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="text" indent="yes" encoding="iso-8859-1"/>
<xsl:template match="/">
/**
 * NTU PDCC SOArDSGrid
 * This class is automatically generated by SOArDSGrid service generator.
 */
package <xsl:value-of select="component/serviceImplPackage"/>;

```

```

import org.globus.wsrf.ResourceKey;
import org.globus.wsrf.impl.ResourceHomeImpl;
import org.globus.wsrf.impl.SimpleResourceKey;

/**
 * @author chen0081
 */
public class <xsl:value-of select="component/resourceHomeClassName"/> extends ResourceHomeImpl {
    public ResourceKey create(String sessionName, String instanceName) throws Exception {
        <xsl:value-of select="component/resourceClassName"/> resource = (<xsl:value-of se-
lect="component/resourceClassName"/>)createNewInstance();
        resource.create(sessionName, instanceName);
        ResourceKey key = new SimpleResourceKey(keyTypeName, resource.getID());
        add(key, resource);
        return key;
    }
}
</xsl:template>
</xsl:stylesheet>

```

C.3 Service Implementation Template

```

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="text" indent="yes" encoding="iso-8859-1"/>
<xsl:template match="/">
package <xsl:value-of select="component/serviceImplPackage"/>;

/**
 * @author chen0081
 * This file is automatically generated by SOAr-DSGrid
 */
import java.rmi.RemoteException;
import java.util.HashMap;

import org.apache.axis.message.addressing.EndpointReferenceType;
import org.globus.wsrf.ResourceContext;
import org.globus.wsrf.ResourceKey;
import org.globus.wsrf.utils.AddressingUtils;

import soardsgrid.component.primitivepull.service.impl.PrimitiveComponent;
import soardsgrid.generated.component.primitivepull.CreateResource;
import soardsgrid.generated.component.primitivepull.CreateResourceResponse;
import soardsgrid.generated.component.primitivepull.Option;

<xsl:variable name="resourceClassName" select="component/resourceClassName"/>

public class <xsl:variable select="component/componentClassName"/> extends PrimitiveComponent {
    /**
     * Called by each operation in PrimitiveComponent to get a component
     * instance to operate on.
     *
     * @return A <xsl:variable name="resourceClassName"/> instance
     * @throws RemoteException
     */
    public <xsl:variable name="resourceClassName"/> getResource() throws RemoteException {
        Object resource = null;
        try {
            resource = ResourceContext.getResourceContext().getResource();
            return (<xsl:variable name="resourceClassName"/>) resource;
        } catch (Exception e) {

```



```

        throw new RemoteException("", e);
    }
}

/**
 * Called by Organizer service or BaseCompositeContainer to create an
 * instance of component resource
 *
 * @return
 * @throws RemoteException
 */
public CreateResourceResponse createResource(CreateResource request) throws RemoteException {
    CreateResourceResponse response = new CreateResourceResponse();
    EndpointReferenceType epr = null;
    try {
        ResourceContext ctx = null;
        <xsl:variable select="component/resourceHomeClassName"/> home = null;
        ResourceKey key = null;
        ctx = ResourceContext.getResourceContext();
        home = (<xsl:variable select="component/resourceHomeClassName"/>)
ctx.getResourceHome();
        // resource home creates resource and returns resource key
        String sessionName = request.getSessionName();
        String instanceName = request.getInstanceName();
        key = home.create(sessionName, instanceName);
        // create endpoint reference
        epr = AddressingUtils.createEndpointReference(ctx, key);
        HashMap optionsMap = new HashMap();
        Option[] options = request.getOptions();
        <![CDATA[
            if(options != null && options.length > 0) {
                int size = options.length;
                for(int i = 0; i < size; i++) {
                    optionsMap.put(options[i].getOptionType(), options[i].getOptionValue());
                }
            }
        ]]>
        <xsl:variable name="resourceClassName"/> resource = (<xsl:variable
name="resourceClassName"/>) home.find(key);
        resource.setOptions(optionsMap);
    } catch (RemoteException e) {
        throw e;
    } catch (Exception e) {
        throw new RemoteException("", e);
    }
    response.setEndpointReference(epr);
    return response;
}
}
</xsl:template>
</xsl:stylesheet>

```

C.4 WSDD Template

```

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="text" indent="yes" encoding="iso-8859-1"/>
<xsl:template match="/">
<![CDATA[
<?xml version="1.0" encoding="UTF-8"?>

```

```

<deployment name="defaultServerConfig"
  xmlns="http://xml.apache.org/axis/wsdd/"
  xmlns:java="http://xml.apache.org/axis/wsdd/providers/java"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:primw="http://soardsgrid/component/primitivePull/PrimitiveComponent.wsdl"
  xmlns:prim="http://soardsgrid/component/primitivePull/PrimitiveComponent.xsd"
  xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing">

  <service name=""]><xsl:value-of select='component/serviceName'/><![CDATA[" provider="Handler"
style="wrapped" use="literal">
  <parameter name="className" value=""]><xsl:value-of se-
lect='component/serviceImplPackage'/>.<xsl:value-of se-
lect='component/componentClassName'/><![CDATA[""/>
  <parameter name="allowedMethods" value="*/>
  <parameter name="handlerClass" value="org.globus.axis.providers.RPCProvider"/>
  <parameter name="scope" value="Application"/>
  <parameter name="providers" value="GetRPPProvider"/>
  <parameter name="loadOnStartup" value="true"/>
</service>

</deployment>
]]>
</xsl:template>
</xsl:stylesheet>

```

C.5 JNDI Template

```

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="text" indent="yes" encoding="iso-8859-1"/>
<xsl:template match="/">
<![CDATA[
<?xml version="1.0" encoding="UTF-8"?>
<jndiConfig xmlns="http://wsrf.globus.org/jndi/config"
xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing">

<service name=""]><xsl:value-of select='component/serviceName'/><![CDATA[">
  <resource name="home" type=""]><xsl:value-of select='component/serviceImplPackage'/>.<xsl:value-
of select='component/resourceHomeClassName'/><![CDATA[">
    <resourceParams>
      <parameter>
        <name>resourceClass</name>
        <value>]]><xsl:value-of select='component/serviceImplPackage'/>.<xsl:value-of se-
lect='component/resourceClassName'/><![CDATA["</value>
      </parameter>
      <parameter>
        <name>factory</name>
        <value>org.globus.wsrf.jndi.BeanFactory</value>
      </parameter>
      <parameter>
        <name>resourceKeyType</name>
        <value>java.lang.String</value>
      </parameter>
      <parameter>
        <name>resourceKeyName</name>
        <value>{]]><xsl:value-of select="component/componentNS"/>}<xsl:value-of se-
lect="component/resourceClassName"/>Key<![CDATA["</value>
      </parameter>
    </resourceParams>

```

```

    </resource>
  </service>
</jndiConfig>
]]>
</xsl:template>
</xsl:stylesheet>

```

C.6 WSDL Template

```

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="text" indent="yes" encoding="iso-8859-1"/>
<xsl:template match="/">

  <xsl:variable name="componentNS" select="component/componentNS"/>
  <xsl:variable name="componentClassName" select="component/componentClassName"/>
  <xsl:variable name="resourceClassName" select="component/resourceClassName"/>

  <![CDATA[
    <?xml version="1.0" encoding="UTF-8"?>
    <wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing"
      xmlns:wsrp="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-draft-01.xsd"
      xmlns:wsrpw="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-draft-
01.wsdl"
      xmlns:wsnt="http://docs.oasis-open.org/wsn/2004/06/wsn-WS-BaseNotification-1.2-draft-01.xsd"
      xmlns:wsntw="http://docs.oasis-open.org/wsn/2004/06/wsn-WS-BaseNotification-1.2-draft-01.wsdl"
      xmlns:wsbf="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-BaseFaults-1.2-draft-01.xsd"
      xmlns:sim="http://soardsgrid/organizer/sim"
      xmlns:prim="http://soardsgrid/component/primitivePull/PrimitiveComponent.wsdl"
      xmlns:primw="http://soardsgrid/component/primitivePull/PrimitiveComponent.wsdl"
      xmlns:primRes="http://soardsgrid/component/primitivePull/PrimitiveResource.xsd"
      xmlns:tns=""]><xsl:variable name='componentNS'/>
      name="<xsl:value-of select='component/componentClassName'/>"
      targetNamespace="<xsl:variable name='componentNS'/><![CDATA[">

    <wsdl:import
      namespace="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-draft-
01.wsdl"
      location="../../../../wsrf/properties/WS-ResourceProperties.wsdl" />
    <wsdl:import
      namespace="http://docs.oasis-open.org/wsn/2004/06/wsn-WS-BaseNotification-1.2-draft-01.wsdl"
      location="../../../../wsrf/notification/WS-BaseN.wsdl" />
    <xsd:import
      namespace="http://soardsgrid/component/primitivePull/PrimitiveComponent.wsdl"
      schemaLocation="PrimitiveComponent.wsdl" />

    <!--
    =====
    ===== Type and Element Definitions =====
    =====
    -->

    <wsdl:types>
      <xsd:schema
        targetNamespace=""]><xsl:variable name='componentNS'/><![CDATA[">

```

```

<!--
=====
===== Imported Schemas =====
=====
-->
<xsd:import namespace="http://soardsgrid/organizer/sim"
  schemaLocation="../../soardsgrid/component/primitivePull/SimulationSchema.xsd" />
<xsd:import
  namespace="http://soardsgrid/component/primitivePull/PrimitiveResource.xsd"
  schemaLocation="../../soardsgrid/component/primitivePull/PrimitiveResource.xsd" />
<xsd:import
  namespace="http://soardsgrid/component/primitivePull/PrimitiveComponent.wsdl"
  schemaLocation="PrimitiveComponent.wsdl" />

<!-- ===== Resource Property Related ===== -->
<xsd:element name="]]><xsl:variable name='componentClassName'/>RP"
  type="tns:<xsl:variable name='resourceClassName'/><![CDATA["/>

<xsd:complexType name="]]><xsl:variable name='resourceClassName'/><![CDATA[">
  <xsd:complexContent>
    <xsd:extension base="prim:PrimitiveComponentResource">
      <xsd:sequence>]]>
        <xsl:for-each select="component/variables/variable">
          <![CDATA[
            <xsd:element name="]]><xsl:value-of select='name'/>" type="xsd:<xsl:value-
of select='type'/><![CDATA["/>]]>
          </xsl:for-each><![CDATA[
            </xsd:sequence>
          </xsd:extension>
        </xsd:complexContent>
      </xsd:complexType>
    </xsd:schema>
  </wsdl:types>

<!--
=====
===== Message Definitions =====
=====
-->

<!--
=====
===== PortType Definitions =====
=====
-->
<wsdl:portType name="]]><xsl:variable name='componentClassName'/>PortType<![CDATA[">
  <wsdl:operation name="CreateResource">
    <wsdl:input message="primw:CreateResourceReqMsg" />
    <wsdl:output message="primw:CreateResourceResMsg" />
  </wsdl:operation>
  <wsdl:operation name="InitComponent">
    <wsdl:input message="primw:InitComponentReqMsg" />
    <wsdl:output message="primw:InitComponentResMsg" />
  </wsdl:operation>
  <wsdl:operation name="Start">
    <wsdl:input message="primw:StartReqMsg" />
    <wsdl:output message="primw:StartResMsg" />
  </wsdl:operation>
  <wsdl:operation name="GetStatus">
    <wsdl:input message="primw:GetStatusReqMsg" />
    <wsdl:output message="primw:GetStatusResMsg" />
  </wsdl:operation>

```

```

<wsdl:operation name="CanReceiveEntity">
  <wsdl:input message="primw:CanReceiveEntityReqMsg" />
  <wsdl:output message="primw:CanReceiveEntityResMsg" />
</wsdl:operation>
<wsdl:operation name="ReceiveEntity">
  <wsdl:input message="primw:ReceiveEntityReqMsg" />
  <wsdl:output message="primw:ReceiveEntityResMsg" />
</wsdl:operation>
<wsdl:operation name="ReceiveMessage">
  <wsdl:input message="primw:ReceiveMessageReqMsg" />
  <wsdl:output message="primw:ReceiveMessageResMsg" />
</wsdl:operation>
<wsdl:operation name="GetSharedVariable">
  <wsdl:input message="primw:GetSharedVariableReqMsg" />
  <wsdl:output message="primw:GetSharedVariableResMsg" />
</wsdl:operation>
<wsdl:operation name="ReceiveVariableUpdate">
  <wsdl:input message="primw:ReceiveVariableUpdateReqMsg"></wsdl:input>
  <wsdl:output message="primw:ReceiveVariableUpdateResMsg"></wsdl:output>
</wsdl:operation>
<wsdl:operation name="Unsubscribe">
  <wsdl:input message="primw:UnsubscribeRequest"></wsdl:input>
  <wsdl:output message="primw:UnsubscribeResponse"></wsdl:output>
</wsdl:operation>
<wsdl:operation name="Stop">
  <wsdl:input message="primw:StopRequest"></wsdl:input>
  <wsdl:output message="primw:StopResponse"></wsdl:output>
</wsdl:operation>
</wsdl:portType>
</wsdl:definitions>
]]>
</xsl:template>
</xsl:stylesheet>

```

APPENDIX D: WAFER FABRICATION FILES

D.1 EtchStrip Component Interface

```

<?xml version="1.0" encoding="UTF-8"?>
<comp:component xmlns:comp="http://www.cbdsg.org/comp"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.cbdsg.org/comp:\project\ComponentInterface.xsd" type="EtchStripTool"
  version="0.1" isComposite="false" provider="chen0081">
  <description>This machine set is in charge of two kinds of processes. 1st: (P6, P15, P24 - Etching). 2nd:
  (P7, P16, P25 - Stripping). ToolSetID = 5, OpSetID = 4 </description>
  <operations>
    <name>Etching</name>
    <description>Remove the oxide where the photoresist pattern is absent</description>
    <inputEntity>
      <type>Wafer</type>
    </inputEntity>
    <outputEntity>
      <type>Wafer</type>
    </outputEntity>
  </operations>
</operations>

```

```

    <name>Stripping</name>
    <description>Strip the photoresist off the wafer completely, leaving the photoresist pattern on
the wafer</description>
    <inputEntity>
      <type>Wafer</type>
    </inputEntity>
    <outputEntity>
      <type>Wafer</type>
    </outputEntity>
  </operations>

  <variables>
    <name>toolSetID</name>
    <type>integer</type>
    <description>ID of the this tool set</description>
    <defaultValue>1</defaultValue>
    <mode>local</mode>
  </variables>

  <variables>
    <name>quantity</name>
    <type>integer</type>
    <description>number of machines in this tool set</description>
    <defaultValue>1</defaultValue>
    <mode>local</mode>
  </variables>

  <variables>
    <name>operatorSetID1</name>
    <type>integer</type>
    <description>ID of the first operator set required for this machine</description>
    <defaultValue>1</defaultValue>
    <mode>local</mode>
  </variables>

  <variables>
    <name>machineSetupTime1</name>
    <type>float</type>
    <description>set up time of machines in this machine set for Etch operation</description>
    <defaultValue>1</defaultValue>
    <mode>local</mode>
  </variables>

  <variables>
    <name>machineSetupTime2</name>
    <type>float</type>
    <description>set up time of machines in this machine set for Strip operation</description>
    <defaultValue>1</defaultValue>
    <mode>local</mode>
  </variables>

  <variables>
    <name>loadTime1</name>
    <type>float</type>
    <description>wafter load time in the first process</description>
    <mode>local</mode>
  </variables>

  <variables>
    <name>processTime1</name>
    <type>float</type>
    <description>wafter process time in the first process</description>
    <mode>local</mode>
  </variables>

```

```

</variables>

<variables>
  <name>unloadTime1</name>
  <type>float</type>
  <description>wafter unload time in the first process</description>
  <mode>local</mode>
</variables>

<variables>
  <name>loadTime2</name>
  <type>float</type>
  <description>wafter load time in the second process</description>
  <mode>local</mode>
</variables>

<variables>
  <name>processTime2</name>
  <type>float</type>
  <description>wafter process time in the second process: epitaxy process</description>
  <mode>local</mode>
</variables>

<variables>
  <name>unloadTime2</name>
  <type>float</type>
  <description>wafter unload time in the second process: epitaxy process</description>
  <mode>local</mode>
</variables>

<variables>
  <name>op1LoadFraction1</name>
  <type>float</type>
  <description>fraction of time first operator is needed to load wafer in the first process on this
machine</description>
  <defaultValue>0</defaultValue>
  <mode>local</mode>
</variables>

<variables>
  <name>op1ProcessFraction1</name>
  <type>float</type>
  <description>fraction of time first operator is needed to process wafer in the first process on
this machine</description>
  <mode>local</mode>
</variables>

<variables>
  <name>op1UnloadFraction1</name>
  <type>float</type>
  <description>fraction of time first operator is needed to unload wafer in the first process on
this machine</description>
  <mode>local</mode>
</variables>

<variables>
  <name>op1LoadFraction2</name>
  <type>float</type>
  <description>fraction of time first operator is needed to load wafer in the second process on
this machine</description>
  <mode>local</mode>
</variables>

```

```

    <variables>
      <name>op1ProcessFraction2</name>
      <type>float</type>
      <description>fraction of time first operator is needed to process wafer in the second process on
this machine</description>
      <mode>local</mode>
    </variables>

    <variables>
      <name>op1UnloadFraction2</name>
      <type>float</type>
      <description>fraction of time first operator is needed to unload wafer in the second process on
this machine</description>
      <mode>local</mode>
    </variables>

    <variables>
      <name>travelTime1</name>
      <type>float</type>
      <description>time spent by wafer to travel to the first possible next machine</description>
      <mode>local</mode>
    </variables>

  </comp:component>

```

D.2 CMP Component Interface

```

<?xml version="1.0" encoding="UTF-8"?>
<comp:component xmlns:comp="http://www.cbdsg.org/comp"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.cbdsg.org/comp:\project\ComponentInterface.xsd" type="CMPTool" ver-
sion="0.1" isComposite="false" provider="chen0081">
  <description>This tool set is in charge of the (P10, P18, P26 - Chemical Mechanical Planarization)
process. ToolSetID = 7, OpSetID = 5</description>
  <operations>
    <name>CMP</name>
    <description>Chemical Mechanical Planarization: polish the wafer smooth</description>
    <inputEntity>
      <type>Wafer</type>
    </inputEntity>
    <outputEntity>
      <type>Wafer</type>
    </outputEntity>
  </operations>

  <variables>
    <name>toolSetID</name>
    <type>integer</type>
    <description>ID of the this tool set</description>
    <defaultValue>1</defaultValue>
    <mode>local</mode>
  </variables>

  <variables>
    <name>quantity</name>
    <type>integer</type>
    <description>number of machines in this tool set</description>
    <defaultValue>1</defaultValue>
  </variables>

```



```

        <mode>local</mode>
    </variables>

    <variables>
        <name>operatorSetID1</name>
        <type>integer</type>
        <description>ID of the first operator set required for this machine</description>
        <defaultValue>1</defaultValue>
        <mode>local</mode>
    </variables>

    <variables>
        <name>machineSetupTime</name>
        <type>float</type>
        <description>set up time of machines in this machine set</description>
        <defaultValue>1</defaultValue>
        <mode>local</mode>
    </variables>

    <variables>
        <name>loadTime1</name>
        <type>float</type>
        <description>wafter load time in the first process</description>
        <defaultValue>0</defaultValue>
        <mode>local</mode>
    </variables>

    <variables>
        <name>processTime1</name>
        <type>float</type>
        <description>wafter process time in the first process</description>
        <defaultValue>0</defaultValue>
        <mode>local</mode>
    </variables>

    <variables>
        <name>unloadTime1</name>
        <type>float</type>
        <description>wafter unload time in the first process</description>
        <defaultValue>0</defaultValue>
        <mode>local</mode>
    </variables>

    <variables>
        <name>op1LoadFraction1</name>
        <type>float</type>
        <description>fraction of time first operator is needed to load wafer in the first process on this
machine</description>
        <defaultValue>0</defaultValue>
        <mode>local</mode>
    </variables>

    <variables>
        <name>op1ProcessFraction1</name>
        <type>float</type>
        <description>fraction of time first operator is needed to process wafer in the first process on
this machine</description>
        <defaultValue>0</defaultValue>
        <mode>local</mode>
    </variables>

    <variables>
        <name>op1UnloadFraction1</name>

```

```

        <type>float</type>
        <description>fraction of time first operator is needed to unload wafer in the first process on
this machine</description>
        <defaultValue>0</defaultValue>
        <mode>local</mode>
    </variables>

</comp:component>

```

D.3 Wafer Fabrication Simulation File

```

<?xml version="1.0" encoding="UTF-8"?>
<sim:simulation sessionName="wafter manufacturing session 0.1">
    <!-- 1st component: ToolSetID = 1 -->
    <componentInstances name="makeWaferTool" comopnentType="MeltonSiliconBathTool" ver-
sion="0.1">
        <setupVariables>
            <name>quantity</name>
            <type>integer</type>
            <value>10</value>
        </setupVariables>
        <setupVariables>
            <name>operatorSetID1</name>
            <type>integer</type>
            <value>1</value>
        </setupVariables>

        <setupVariables>
            <name>machineSetupTime</name>
            <type>float</type>
            <value>10.0</value>
        </setupVariables>

        <setupVariables>
            <name>loadTime1</name>
            <type>float</type>
            <value>1.0</value>
        </setupVariables>

        <setupVariables>
            <name>processTime1</name>
            <type>float</type>
            <value>720.0</value>
        </setupVariables>

        <setupVariables>
            <name>unloadTime1</name>
            <type>float</type>
            <value>10.0</value>
        </setupVariables>

        <setupVariables>
            <name>loadTime2</name>
            <type>float</type>
            <value>10.0</value>
        </setupVariables>

        <setupVariables>

```

```

        <name>processTime2</name>
        <type>float</type>
        <value>720.0</value>
    </setupVariables>
    <setupVariables>
        <name>unloadTime2</name>
        <type>float</type>
        <value>10.0</value>
    </setupVariables>

    <setupVariables>
        <name>op1LoadFraction1</name>
        <type>float</type>
        <value>0.25</value>
    </setupVariables>
    <setupVariables>
        <name>op1ProcessFraction1</name>
        <type>float</type>
        <value>0.1</value>
    </setupVariables>
    <setupVariables>
        <name>op1UnloadFraction1</name>
        <type>float</type>
        <value>0.25</value>
    </setupVariables>

    <setupVariables>
        <name>op1LoadFraction2</name>
        <type>float</type>
        <value>0.25</value>
    </setupVariables>
    <setupVariables>
        <name>op1ProcessFraction2</name>
        <type>float</type>
        <value>0.2</value>
    </setupVariables>
    <setupVariables>
        <name>op1UnloadFraction2</name>
        <type>float</type>
        <value>0.25</value>
    </setupVariables>
    <setupVariables>
        <name>travelTime1</name>
        <type>float</type>
        <value>1.0</value>
    </setupVariables>
</componentInstances>

<!-- 2nd component: ToolSetID = 2 -->
<componentInstances name="makeMaskTool" componentType="MakeMaskTool" version="0.1">
    <setupVariables>
        <name>quantity</name>
        <type>integer</type>
        <value>10</value>
    </setupVariables>
    <setupVariables>
        <name>operatorSetID1</name>
        <type>integer</type>
        <value>1</value>
    </setupVariables>

    <setupVariables>
        <name>machineSetupTime</name>

```

```

    <type>float</type>
    <value>10.0</value>
</setupVariables>

<setupVariables>
    <name>loadTime1</name>
    <type>float</type>
    <value>1.0</value>
</setupVariables>

<setupVariables>
    <name>processTime1</name>
    <type>float</type>
    <value>720.0</value>
</setupVariables>
<setupVariables>
    <name>unloadTime1</name>
    <type>float</type>
    <value>10.0</value>
</setupVariables>
<setupVariables>
    <name>loadTime2</name>
    <type>float</type>
    <value>10.0</value>
</setupVariables>
<setupVariables>
    <name>processTime2</name>
    <type>float</type>
    <value>720.0</value>
</setupVariables>
<setupVariables>
    <name>unloadTime2</name>
    <type>float</type>
    <value>10.0</value>
</setupVariables>
<setupVariables>
    <name>op1LoadFraction1</name>
    <type>float</type>
    <value>0.25</value>
</setupVariables>
<setupVariables>
    <name>op1ProcessFraction1</name>
    <type>float</type>
    <value>0.1</value>
</setupVariables>
<setupVariables>
    <name>op1UnloadFraction1</name>
    <type>float</type>
    <value>0.25</value>
</setupVariables>

<setupVariables>
    <name>op1LoadFraction2</name>
    <type>float</type>
    <value>0.25</value>
</setupVariables>
<setupVariables>
    <name>op1ProcessFraction2</name>
    <type>float</type>
    <value>0.2</value>
</setupVariables>
<setupVariables>
    <name>op1UnloadFraction2</name>

```

```

        <type>float</type>
        <value>0.25</value>
    </setupVariables>
    <setupVariables>
        <name>travelTime1</name>
        <type>float</type>
        <value>1.0</value>
    </setupVariables>
</componentInstances>

<!-- 3rd component: ToolSetID = 3 -->
<componentInstances name="oxidationTool" componentType="OxidationTool" version="0.1">
    ...
</componentInstances>

<!-- 4th component: ToolSetID = 4 -->
<componentInstances name="stepper" componentType="PhotolithographyTool" version="0.1">
    ...
</componentInstances>

<!-- 5th component: ToolSetID = 5 -->
<componentInstances name="etchStripMachine" componentType="EtchStripMachine" ver-
sion="0.1">
    ...
</componentInstances>

<!-- 6th component: ToolSetID = 6 -->
<componentInstances name="diffusionImplantationMachine" component-
Type="DiffusionImplantationMachine" version="0.1">
    ...
</componentInstances>

<!-- 7th component: ToolSetID = 7 -->
<componentInstances name="depositionTool" componentType="DepositionTool" version="0.1">
    ...
</componentInstances>

<!-- 8th component: ToolSetID = 8 -->
<componentInstances name="matalizationMachine" componentType="MatalizationMachine" ver-
sion="0.1">
    ...
</componentInstances>

<!-- 9th component: ToolSetID = 9 -->
<componentInstances name="aCMPMachine" componentType="CMPMachine" version="0.1">
    ...
</componentInstances>

<!-- 10th component: ToolSetID = 10 -->
<componentInstances name="tester" componentType="Tester" version="0.1">
    ...
</componentInstances>

<composition>
    <entityFlows entityType="Wafer">
        <entityLinks>
            <!-- link from ToolSetID 1 to ToolSetID 2 -->
            <source>makeWafer</source>
            <targets>
                <target>makeMaskTool</target>
                <travelTime>1.0</travelTime>
            </targets>
        </entityLinks>
    </entityFlows>

```

```

<entityLinks>
  <!-- link from ToolSetID 2 to ToolSetID 3 -->
  <source>makeMaskTool</source>
  <targets>
    <target>oxidationTool</target>
    <travelTime>1.0</travelTime>
  </targets>
</entityLinks>

<entityLinks>
  <!-- link from ToolSetID 3 to ToolSetID 4 -->
  <source>oxidationTool</source>
  <targets>
    <target>stepper</target>
    <travelTime>1.0</travelTime>
  </targets>
</entityLinks>

<entityLinks>
  <!-- link from ToolSetID 4 to ToolSetID 5 -->
  <source>stepper</source>
  <targets>
    <target>etchStripMachine</target>
    <travelTime>1.0</travelTime>
  </targets>
</entityLinks>

<entityLinks>
  <!-- link from ToolSetID 5 to ToolSetID 6 -->
  <source>etchStripMachine</source>
  <targets>
    <target>diffusionImplantMachine</target>
    <travelTime>1.0</travelTime>
  </targets>
</entityLinks>

<entityLinks>
  <!-- link from ToolSetID 6 to ToolSetID 7 -->
  <source>diffusionImplantMachine</source>
  <targets>
    <target>depositionTool</target>
    <travelTime>1.0</travelTime>
  </targets>
</entityLinks>

<entityLinks>
  <!-- link from ToolSetID 7 to ToolSetID 8 -->
  <source>depositionTool</source>
  <targets>
    <target>metalizationMachine</target>
    <travelTime>1.0</travelTime>
  </targets>
</entityLinks>

<entityLinks>
  <!-- link from ToolSetID 8 to ToolSetID 9 -->
  <source>metalizationMachine</source>
  <targets>
    <target>aCMPMachine</target>
    <travelTime>1.0</travelTime>
  </targets>
</entityLinks>

```

```

    <entityLinks>
      <!-- link from ToolSetID 9 to ToolSetID 10 or ToolSetID 3 -->
      <source>aCMPMachine</source>
      <targets>
        <target>oxidationTool</target>
        <travelTime>1.0</travelTime>
      </targets>
      <targets>
        <target>tester</target>
        <travelTime>1.0</travelTime>
      </targets>
      <routingRule applyToEntity="Wafer" name="CMPRoutingRule"
type="LessThanBranch">
        <configVars name="variable1">
          <objectName>currentEntity</objectName>
          <attributeName>loopCount</attributeName>
        </configVars>
        <configVars name="variable2">
          <type>integer</type>
          <value>3</value>
        </configVars>

        <configVars name="exit1">
          <type>string</type>
          <value>oxidationTool</value>
        </configVars>
        <configVars name="exit2">
          <type>string</type>
          <value>tester</value>
        </configVars>
      </routingRule>
    </entityLinks>

  </entityFlows>
  <variableFlows>
    <provider>
      <providerName>oxidationTool</providerName>
      <variableName>isInputQueueFull</variableName>
    </provider>
    <subscribers>
      <subscriberName>makeMaskTool</subscriberName>
      <variableName>isOxInputQueueFull</variableName>
    </subscribers>
    <subscribers>
      <subscriberName>aCMPMachine</subscriberName>
      <variableName>isOxInputQueueFull</variableName>
    </subscribers>
  </variableFlows>
</composition>
<results>
  <status>
    <componentName>makeWaferTool</componentName>
    <variableName>numOfWafers</variableName>
    <variableType>integer</variableType>
  </status>
  <status>
    <componentName>tester</componentName>
    <variableName>rejectRate</variableName>
    <variableType>float</variableType>
  </status>
</results>
</sim:simulation>

```

APPENDIX E : GRAPHIC USER INTERFACES

E.1 Service Generator GUI

The image shows a screenshot of the SOArDSGrid Test Suite GUI. The window title is "SOArDSGrid Test Suite". A green message bar at the top states: "Generate service successfully. You may proceed to build the service." Below this, there are three tabs: "Generator", "Index", and "Organizer". The "Generator" tab is active. It contains several input fields for configuration:

- Component Interface File Path: D:/soardsgridRepo/share/schema/soardsgrid/CompA.xml
- Intermediate XML Path: D:/soardsgridRepo/share/schema/soardsgrid/CompAIntermediateXML.xml
- ServiceGen XML Path: D:/soardsgridRepo/share/schema/soardsgrid/CompAServiceGen.xml
- Gar ID: soardsgrid_component_proto1_compA_service_CompA
- Gar File Path: D:/soardsgridRepo/temp/soardsgrid_component_proto1_compA_service_CompA.gar
- Service Name: ReaderComp
- Service URL: http://127.0.0.1:8080/wsrf/services/soardsgrid/proto1/CompA
- Shutdown Service URL: http://127.0.0.1:8080/wsrf/services/ShutdownService

On the right side of the GUI, there are seven buttons arranged vertically:

- Generate Service
- Build Service
- Undeploy Service
- Deploy Service
- Register Service
- Start Globus Container
- Stop Globus Container

E.2 Index Service GUI

The screenshot shows the 'SOArDSGrid Test Suite' application window with the 'Index' tab selected. At the top, a status bar indicates 'Index information retrieved.' Below this, there are three tabs: 'Generator', 'Index', and 'Organizer'. The 'Index' tab contains several input fields and a table. On the right side of the tab, there are five buttons: 'Browse', 'Is Service Registered', 'Retrieve Service URL', 'Register Service', and 'Deregister Service'.

IndexService URL:

Service Name:

Service URL:

ServiceName	ServiceURL
OwnerComp	http://127.0.0.1:8080/wsrf/services/soardsgrid/proto1/CompA
ReaderComp	http://127.0.0.1:8080/wsrf/services/soardsgrid/proto1/CompB

E.3 Organizer Service GUI

APPENDIX F : GRID METHODS DESCRIPTION

F.1 PrimitiveComponent Grid Methods Description

<i>createResource()</i> :	to create a new resource instance and return the endpoint reference to the resource instance to the caller.
<i>initComponent()</i> :	to initialize a component instance by the following steps. setting all the setup variables, configuring the routing rule, initializing the linked components information such as endpoint references of linked components, names of preceding components, names of following components and travel time to those components, names of the provider components and their corresponding shared variables, names of the consumers components and their corresponding subscribed variables.
<i>subscribeStatus()</i> :	Used by the organizer to subscribe the termination status of a component resource.
<i>start()</i> :	to trigger an simulation program associated with current component resource.
<i>getStatus()</i> :	to check the execution status of the current component.
<i>stop()</i> :	to stop the current executing component instance.
<i>canReceiveEntity()</i> :	to probe whether the external entity queue of a component resource is not full.
<i>receiveEntity()</i> :	by calling this method, the caller component instance sends an entity to the callee component instance. The callee inserts the entity into the input queue according to the ascending order of the timestamp of the entity.
<i>getSharedVariable()</i> :	invoked by the consumer components to retrieve the shared variable value from current component.
<i>receiveVariableUpdate()</i> :	invoked by the provider components of current component to update shared variables asynchronously in an unsolicited manner.
<i>updateTimeAdvancement()</i> :	invoked by the preceding components to update their simulation time with the current component.
<i>unsubscribe()</i> :	invoked by the consumer components to unsubscribe a shared variable. This could relieve the current component from some unnecessary unsolicited variable update.

F.2 Organizer Grid Methods Description

<i>createResource()</i> :	to create a new resource instance and return the endpoint reference to the resource instance to the caller.
<i>submit()</i> :	to submit a simulation description file to the organizer. The parameter is a in memory String representation of the simulation description file.
<i>start()</i> :	to start a simulation session. This will signal the organizer resource to start parsing and transforming the submitted simulation descrpton file, creating simulation session, and triggering the simulation of each component resource.
<i>stop()</i> :	to stop the simulation prematurely. This will cause the organizer to stop simulation execution of all the component resources in the simulation session.
<i>getStatus()</i> :	to check the execution status of a specified component resource. The organizer will invoke the <i>getStatus()</i> grid method of the corresponding component resource for the status.
<i>notifyStatus()</i> :	to notify the organizer of the termination status of a component resource. This is the only grid method for component resources. This method will be invoked when a component resource terminates its simulation execution.

PUBLICATIONS

1. 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.
2. 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.
3. 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.

BIBLIOGRAPHY

- [1] W. M. P. v. d. Aalst, M. Dumas, A. H. M. d. Hofstede, Pattern Based Analysis of BPEL4WS. <http://xml.coverpages.org/AalstBPEL4WS.pdf>.
- [2] W. M. P. v. d. Aalst, M. Dumas, A. H. M. t. Hofstede, Web Service Composition Languages: Old Wine in New Bottles?, in *29th Euromicro Conference (EUROMICRO'03)*, pp. 298-305, September 2003.
- [3] AFWs, Application Factory Web Service homepage. <http://www.extreme.indiana.edu/xgws/afw/>.
- [4] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, B. Smolinski, Toward a Common Component Architecture for High-Performance Scientific Computing, in *Proceedings of High Performance Distributed Computing Conference*, pp. 13-22, 1999.
- [5] BOM, Basic Object Model. <http://www.boms.info/>.
- [6] BPEL4WS, Business Process Execution Language for Web Services version 1.1. <http://www-128.ibm.com/developerworks/library/specification/ws-bpel/>.
- [7] R. J. Brunner, F. Cohen, F. Curbera, D. Govoni, S. Haines, M. Kloppmann, B. Marchal, S. Morrison, A. Ryman, J. Weber, M. Wutka, Web Services and Flows (WSFL), in *Java Web Services Unleashed*. Sams Publishing, 2002.
- [8] A. Buss, Component-based Simulation Modeling, in *Proceedings of the 2000 Winter Simulation Conference*, Orlando, USA, pp. 964-971, 2000.
- [9] A. Buss, Component Based Simulation Modeling with SimKit, in *Proceedings of the 2002 Winter Simulation Conference*, San Diego, USA, pp. 243-249, 2002.
- [10] A. Buss, L. Jackson, Distributed Simulation Modeling: A Comparison of HLA, CORBA, and RMI, in *Proceedings of the 1998 Winter Simulation Conference*, pp. 819-825, 1998.
- [11] A. H. Buss, P. J. Sanchez, Building Complex Models with LEGOs, in *Proceedings of the 2002 Winter Simulation Conference*, San Diego, USA, pp. 732-737, 2002.
- [12] W. Cai, S. J. Turner, B. P. Gan, Hierarchical Federations: An Architecture for Information Hiding, in *Proceedings of 15th Workshop on Parallel and Distributed Simulation*, pp. 67-74, 2001.
- [13] CCA, The Common Component Architecture Forum. <http://www.cca-forum.org/glossary/index.html>.
- [14] Ccaffeine, Ccaffeine Homepage. <http://www.cca-forum.org/ccafe/>.
- [15] CCASpec, The Common Component Architecture Technical Specification - Version 0.5. <http://www.cca-forum.org/bindings/old-0.5/>.
- [16] CCM, CORBA Component Model. http://www.omg.org/technology/documents/corba_spec_catalog.htm#CCM.
- [17] Y. I. Cho, T. G. Kim, DEVS Framework for Component-based Modeling/Simulation of Discrete Event System. <http://www.scs.org/scsarchive/getDoc.cfm?id=1777>.
- [18] COM, COM: Component Object Model Technologies. <http://www.microsoft.com/com/default.mspx>.
- [19] CORBA, OMG CORBA Interface Definition Language (IDL). <http://www.objs.com/x3h7/corbaidl.htm>.
- [20] CORBA, Overview of CORBA. <http://www.cs.wustl.edu/~schmidt/corba-overview.html>.
- [21] K. Czajkowski, D. Ferguson, I. Foster, J. Graham, T. Maguire, D. Snelling, S. Tuecke, From Open Grid Services Infrastructure to WS-Resource Framework: Refactoring and Evolution. http://www-128.ibm.com/developerworks/library/ws-resource/ogsi_to_wsrf_1.0.pdf.
- [22] DMSO, DMSO RTI. <https://www.dmsol.com/public/transition/hla/rti/>.
- [23] EJB, Enterprise JavaBeans Technology. <http://java.sun.com/products/ejb/>.

MASTER THESIS

BIBLIOGRAPHY

- [24] M. Farshad, Component-Based Simulation Model Development Using BOMs and Web Services, in *Proceedings of the First Asia International Conference on Modelling and Simulation (AMS 2007)*, Phuket, Thailand, pp. 238-246, 2007.
- [25] M. Farshad, N. Peder, A. Rassul, Simulation Model Composition using BOMs, in *Proceedings of the 10th IEEE international symposium on Distributed Simulation and Real-Time Applications*, Terremolinos, Spain, pp. 242-252, 2006.
- [26] M. Farshad, A. Rassul, T. Gary, A Rule-based Approach to Syntactic and Semantic Composition of BOMs, in *Proceedings of the 11th IEEE International Symposium on Distributed Simulation and Real-Time Applications*, Chania, Greece, pp. 145-155, 2007.
- [27] M. Farshad, A. Rassul, M. Imran, An Agent-Based Environment for Simulation Model Composition, in *Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation*, Rome, Italy, pp. 175-184, 2008.
- [28] FDK, Federated Simulations Development Kit. <http://www.cc.gatech.edu/computing/pads/fdk/>.
- [29] G. Feigin, J. Fowler, R. Leachman, Semiconductor Wafer Manufacturing Data Format Specification.
- [30] I. Foster, C. Kesselman, Concepts and Architecture, in *The Grid 2: Blueprint for a New Computing Infrastructure*, pp. 37-64. Morgan Kaufman, 2004.
- [31] I. Foster, C. Kesselman, S. Tuecke, The Anatomy of the Grid: Enabling Scalable Virtual Organizations, in *International Journal of Supercomputing Applications*, 15(3), pp. 200-222, 2002.
- [32] I. Foster, H. Kishimoto, A. Savva, The Open Grid Service Architecture, Version 1.0. <http://www.gridforum.org/documents/GWD-I-E/GFD-I.030.pdf>.
- [33] R. M. Fujimoto, Conservative Synchronization Algorithms, in *Parallel and Distributed Simulation Systems*, pp. 52-60. Wiley Interscience, 2000.
- [34] R. M. Fujimoto, Parallel Discrete Event Simulation, in *Communication of the ACM*, 33(10), pp. 30-53, 1990.
- [35] R. M. Fujimoto, Zero Lookahead and Repeatability in the High Level Architecture, in *1997 Spring Simulation Interoperability Workshop*, Orlando, Florida, United States, pp. 3-7, 1997.
- [36] D. Gannon, R. Ananthakrishnan, S. Krishnan, M. Govindaraju, L. Rama-krishnan, A. Slominski, Grid Web Services and Application Factories, in *Grid Computing: Making the Global Infrastructure a Reality*. 2002.
- [37] GGF, Global Grid Forum. <http://www.ggf.org/>.
- [38] Globus, Globus Toolkit. <http://www.globus.org/toolkit/>.
- [39] M. Govindaraju, S. Krishnan, K. Chiu, A. Slominski, D. Gannon, R. Bramley, Merging the CCA Component Model with the OGSF Framework, in *Proceedings of 3rd International Symposium on Cluster Computing and the Grid*, Tokyo, Japan, pp. 182-199, 2003.
- [40] G. T. Heineman, W. T. Councill, *Component-Based Software Engineering: Putting the Pieces Together*, Addison Wesley, 2001.
- [41] HLA, High Level Architecture. <https://www.dmsomil/public/transition/hla/>.
- [42] R. C. Hofer, M. L. Loper, DIS Today, in *Proceedings of the IEEE*, pp. 1124-1137, 1995.
- [43] IEEE, IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Federate Interface Specification.
- [44] IEEE, IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Framework and Rules.
- [45] IEEE, IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Object Model Template (OMT) Specification.
- [46] IEEE, IEEE Standard Glossary of Modeling and Simulation Terminology.
- [47] INFRASTRUCTURE, Semiconductor Manufacturing Tour. <http://www.infras.com/Tutorial/>.

MASTER THESIS

BIBLIOGRAPHY

- [48] I. Jacobson, M. Christerson, P. Jonsson, G. Overgaard, *Object-oriented Software Engineering: a Use Case Driven Approach*, ACM Press, 1992.
- [49] S. Krishnan, D. Gannon, XCAT3: A Framework for CCA Components as OGSA Services, in *Proceedings of 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, Santa Fe, USA, pp. 90-97, 2004.
- [50] S. Krishnan, P. Wagstrom, G. v. Laszewski, GSFL: A Workflow Framework for Grid Services, Argonne National Laboratory, August 2002.
- [51] J. Ledin, in *Simulation Engineering - Build Better Embedded Systems Faster*, pp. 67-74. CMP Books, 2001.
- [52] C. C. Lim, Y. H. Low, B. P. Gan, S. Jain, Implementation of Dispatch Rules in Parallel Manufacturing Simulation, in *Proceedings of 1998 Winter Simulation Conference*, Washington, USA, pp. 1591-1597, 1998.
- [53] Y. H. Low, B. P. Gan, J. Wei, X. Wang, S. J. Turner, W. Cai, Implementation Issues for Shared State in HLA-Based Distributed Simulation, in *the 15th European Simulation Symposium*, Delft, The Netherlands, pp. 5-13, 2003.
- [54] L. Malcolm Yoke Hean, G. Boon Ping, W. Junhu, W. Xiaoguang, T. Stephen John, C. Wentong, Shared State Synchronization for HLA-Based Distributed Simulation, in *Simulation: Transactions of the Society for Modeling and Simulation International*, 82(8), pp. 511-521, 2006.
- [55] B. McWhirter, M. Proctor, Drools (JBoss Rules). <http://en.wikipedia.org/wiki/Drools>.
- [56] H. Mehl, S. Hammes, Shared Variables in Distributed Simulation, in *7th Workshop on Parallel and Distributed Simulation (PADS'93)*, San Diego, California, United States, pp. 68-75, 1993.
- [57] D. C. Miller, J. A. Thorpe, SIMNET: The Advent of Simulator Networking, in *Proceedings of the IEEE*, pp. 1114-1123, 1995.
- [58] Pitch, pRTI. <http://www.pitch.se/prti/default.asp>.
- [59] H. Praehofer, J. Sameting, A. Stritzinger, Component frameworks - A Case Study, in *Proceedings of TOOLS 30 conference*, Santa Barbara, USA, pp. 148-157, 1999.
- [60] H. Praehofer, J. Sameting, A. Stritzinger, Concepts and Architecture of a Simulation Framework Based on the JavaBeans Component Model. <http://www.swe.uni-linz.ac.at/publications/pdf/TR-SE-99.06.pdf>.
- [61] S. W. Reichenthal, SRML - Simulation Reference Markup Language. <http://www.w3.org/TR/2002/NOTE-SRML-20021218/>.
- [62] RPC, Remote Procedure Call (RPC). <http://www.sei.cmu.edu/str/descriptions/rpc.html>.
- [63] SOAP, Simple Object Access Protocol (SOAP) 1.2. <http://www.w3.org/TR/soap>.
- [64] C. Szyperski, Beyond Objects: Point, Counterpoint. <http://www.sdmagazine.com/documents/s=825/sdm00021/>.
- [65] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maquire, T. Sandholm, D. Snelling, P. Vanderbilt, Open Grid Services Infrastructure (OGSI) Version 1.0. http://www-unix.globus.org/toolkit/draft-ggf-ogsi-gridservice-33_2003-06-27.pdf.
- [66] S. J. Turner, W. Cai, C. C. Lin, Y. H. Low, W. J. Hsu, S. Y. Huang, A Methodology for Automating the Parallelization of Manufacturing Simulation, in *the 12th Workshop on Parallel and Distributed Simulation*, Banff, Alberta, Canada, pp. 126-133, 1998.
- [67] Y. Wang, A framework for component-based distributed simulation on the grid: execution support, Nanyang Technological University (2008).
- [68] Y. Wang, S. J. Turner, W. Cai, X. Chen, Time management in a service-oriented architecture for distributed simulation on the grid, in *Proceedings of the 2007 summer computer simulation conference*, San Diego, USA, pp. 392-399, 2007.
- [69] Y. H. Wang, S. H. Ho, Implementation of a DEVS-JavaBean simulation environment, in *Proceedings of Simulation Symposium*, pp. 333-338, 2001.
- [70] J. Wbber, M. Little, Introducing BP4WS 1.0. <http://www.sys-con.com/webservices/>.

MASTER THESIS

BIBLIOGRAPHY

- [71] R. M. Weatherly, ALSP - Theory, Experience, and Future Directions, in *Proceedings of the 1993 Winter Simulation Conference*, Los Angeles, USA, pp. 1068-1072, 1993.
- [72] S. Weerawarana, F. Curbera, Business Processes: Understanding BPEL4WS, Part 1. <http://www-106.ibm.com/developerworks/webservices/library/ws-bpelcoll/>.
- [73] A. L. Wilson, R. M. Weatherly, The Aggregate Level Simulation Protocol: An Evolving System, in *Proceedings of the 1994 Winter Simulation Conference*, Orlando, USA, pp. 781-787, 1994.
- [74] WSA, Web Service Architecture. <http://www.w3.org/TR/ws-arch/>.
- [75] WSDL, Web Service Definition Language (WSDL). <http://www.w3.org/TR/wsdl>.
- [76] WSFL, Web Services Flow Language (WSFL 1.0). <http://www-306.ibm.com/software/solutions/webservices/pdf/WSFL.pdf>.
- [77] WSRF, The Web-Service Resource Framework (WSRF) 1.0. <http://www.globus.org/wsrf/specs/ws-wsrf.pdf>.
- [78] WSRF.NET, WSRF.NET. <http://www.cs.virginia.edu/~gsw2c/wsrf.net.html>.
- [79] WSRF::Lite, WSRF::Lite - An Implementation of the Web Services Resource Framework. <http://www.sve.man.ac.uk/Research/AtoZ/ILCT>.
- [80] XCAT, XCAT homepage. <http://www.extreme.indiana.edu/xcat/index.html>.
- [81] XMLSchema, XML Schema Primer. <http://www.w3.org/TR/xmlschema-0/>.
- [82] B. P. Zeigler, D. Kim, S. J. Buckley, Distributed supply chain simulation in a DEVS/CORBA execution environment, in *Proceedings of the Winter Simulation Conference*, pp. 1333-1340, 1999.
- [83] B. P. Zeigler, H. Praehofer, T. G. Kim, *Theory of Modeling and Simulation, 2nd edition*, Academic Press, 2000.
- [84] K. Zhang, K. Damevski, V. Venkatachalapathy, S. Parker, SCIRun2: A CCA Framework for High Performance Computing, in *Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, Santa Fe, USA, pp. 72-79, 2004.