INTEROPERABILITY BETWEEN DEVS SIMULATORS USING SERVICE ORIENTED ARCHITECTURE AND DEVS NAMESPACE

by

Chungman Seo

A Dissertation Submitted to the Faculty of the

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfillment of the Requirements

For the Degree of

DOCTOR OF PHILOSOPHY

In the Graduate College

THE UNIVERSITY OF ARIZONA

2009

THE UNIVERSITY OF ARIZONA
GRADUATE COLLEGE

As members of the Dissertation Committee, we certify that we have read the dissertation prepared by Chungman Seo
entitled Interoperability between DEVS Simulators using Service Oriented Architecture and DEVS Namespace
and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy in Electrical and Computer Engineering.

____ Date: 03/27/09
Bernard P. Zeigler

____ Date: 03/27/09
Jonathan Sprinkle

____ Date: 03/2709
Ali Akoglu

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copies of the dissertation to the Graduate College.
I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

____ Date: 03/27/09
Dissertation Director: Bernard P. Zeigler

ACKNOWLEDGEMENTS

I would like to express my greatest appreciation to my advisor Dr. Bernard P. Zeigler, who guided me through this research work and introduced me to many exciting areas in discrete event simulation and its application. His help and support are endless. He made me gain new knowledge and insights in my career, without him I would never reach to this point. I am very grateful for his dedication and advising.

I also express my appreciations to the committee members Dr. Jonathan Sprinkle and Dr. Ali Akoglu for providing suggestions enhancing the content of this dissertation.

I would like to express thank to my colleagues at ACIMS Lab, Ho Jun Lee, Lahiru Ariyananda, Dr. Han, and Dr. DH Kim for the useful discussion and advice.

I would like to thank to my parents, sisters, and brothers who always support and trust me.

Special thank goes to my wife, Jeongyeon, for her endless support and love. I can not make it without her. She is everything to me.

TABLE OF CONTENTS

TABLE OF CONTENTS - Continued

TABLE OF CONTENTS - Continued

TABLE OF CONTENTS - Continued

LIST OF ILLUSTRATIONS

LIST OF TABLES

ABSTRACT

Interoperability between heterogeneous software systems is an important issue to increase software reusability in the software industry. Many methods are proposed to implement interoperable systems using distributed computing infrastructures such as CORBA, HLA and SOA. Those infrastructures can provide communication channels between software systems with heterogeneous environments. SOA (Service Oriented Architecture) provides a more flexible approach to interoperability than do the others because it provides platform independence and employs platform-neutral message passing with Simple Object Access Protocol (SOAP) to communicate between a service and a client.

The main contribution of this study is to design and implement an interoperable DEVS simulation environment using the SOA concept and a new construct called the DEVS namespace. The interoperable DEVS environment consists of a DEVS simulator service and an associated integrator. The DEVS simulator service provides both simulator level and model level interoperability. Moreover, using the DEVS namespace, DEVS simulator services can be interoperable with any services using the same message types.

To demonstrate the utility of the proposed environment, we describe various applications of the interoperable DEVS simulation environment. The applications are drawn from real world development of automated testing environments for military information system interoperability. A radar track generation and display federation and a model negotiation web service illustrated the ability of the proposed middleware to work

across platforms and languages. Its ability to support higher level semantic interoperability will be demonstrated in a testing service that can deploy model agents to provide coordinated observation of web requests of participants in simulated distributed scenarios.

CHAPTER 1. INTRODUCTION

1.1. Motivation and Goals

The study of interoperability has been conducted to suggest a methodology to integrate different systems distributed over the network systems. The integrated system called the System of Systems (SoS) is differentiated from a single monolithic system in that it requires interoperability among its constituent systems [1]. SoS engineering has priority on interoperability on the development of command and control (C2) capabilities for joint and coalition warfare [2-4]. From the research of interoperability, models with levels of interoperability describing technical interoperability and the complexity of interoperations [5-7] are suggested in the SoS research groups. The model of levels of interoperability is reinterpreted in the different applications such as telecommunication and software to search their own interoperability levels.

As a result, [8] introduced linguistic levels of interoperability divided into three levels: pragmatic level, semantic level, and syntactic level. The pragmatic level stresses data used in relation to data structure and context of application. The semantic level has a low level focusing on definitions and attributes of terms, and a high level focusing on the combined meaning of multiple terms. The syntactic level focuses on a structure and adherence to the rules that govern that structure. The linguistic levels interoperability concept provides a simultaneous testing environment at multiple levels.

Interoperability between heterogeneous software systems is an important issue to increase software reusability in the software industry. Many methods are proposed to implement interoperable systems using distributed computing infrastructures such as CORBA, HLA and SOA [9-11]. Those infrastructures can provide communication channels between software systems with heterogeneous environments. SOA (Service Oriented Architecture) provides a more flexible approach to interoperability than others because it provides platform independence and employs neutral message passing with Simple Object Access Protocol (SOAP) to communicate between a service and a client [12-15].

The research groups of DEVS modeling and simulation have been interested in interoperable DEVS modeling and simulation in order to enhance model composability and reusability with DEVS models and non DEVS models in different languages and platforms. The problem to interoperate heterogeneous DEVS models with DEVS simulators is that DEVS simulators implement the DEVS modeling formalism in diverse programming environments (e.g. DEVSJAVA, ADEVS, PythonDEVS) [27, 28]. Though the DEVS formalism specifies the same abstract simulator algorithm for any simulator, different simulators implement the same abstract simulator using different codes. This situation inhibits interoperating DEVS simulators and prevents simulation of heterogeneous models. Also, each simulator can not provide platform-neutral message passing.

The interoperable DEVS simulation has been tried to develop the interoperable framework through DEVS standard to simulate DEVS models generated in the different languages and platforms. Some research of interoperability on DEVS has been studied along with HLA and SOA [10, 11]. Prior work includes DEVS/SOA which Mittal and Rico developed using web services [11]. However, it provides only platform interoperability because it employs JAVA serialization which converts JAVA objects into byte array to send messages to simulators. This restricts interoperation to simulators based on JAVA. To add the language interoperability to the platform interoperability, we apply neutral message passing and the SOA environment. The interoperability on DEVS uses simulator level interoperability that uses common simulator interfaces to simulate DEVS models. The simulator interface describes a minimum agreement being able to implement a simulator class using different languages such as JAVA, C++, and C#. This approach strengthens model reusability because DEVS modeling and simulation separates models and simulators. To increase model composibility, we apply a new construct called the DEVS namespace which is a specific XML namespace to define unique message types used at DEVS models in the DEVS simulator services. It provides semantic interoperability when we integrate different DEVS simulators.

The main contribution of this study is to design and implement interoperable DEVS simulation environment using SOA and DEVS namespace. The interoperable DEVS simulation environment is categorized to the design of DEVS simulator service and DEVS simulator service integrator. The DEVS simulator service provides not only simulator level interoperability, but also model level interoperability. Also, through the DEVS namespace, we can couple DEVS simulator services with same message types. In an interoperable DEVS environment, web services represent DEVS simulators embedding specific DEVS models. They have minimum agreement for simulator and information of input/output ports which have specific data types described in DEVS namespace.

1.2. Organization of the Thesis

Background knowledge, discrete event system modeling and simulation, SOA, and interoperability studies are discussed in the chapter 2. Chapter 3 addresses the overall system architecture of DEVS simulator service interoperability consisting of system of interoperability of DEVS simulator services, DEVS namespace, and DEVS simulation service integration and execution. Chapter 4 explains implementation of DEVS namespace and DEVS simulator services. In chapter 4 we demonstrate two DEVS simulator services using JAVA and VC++ with DEVSJAVA and ADEVS, respectively. The example of integration of DEVS simulator services is presented in section 4. In chapter 5, we present application of interoperability of DEVS simulator services. The track display and negotiation systems are integrated among DEVS simulator services implemented with different languages. The testing agents system is implemented using DEVSJAVA modeling and simulation and DEVS simulator services with real time simulator. In chapter 6, we discuss the difference between concept level and implementation level in DEVS simulator service, as well as some issues about web service and platform. The paper’s summary and future work are presented in chapter 7.

CHAPTER 2. BACKGROUND

2.1. Discrete Event System Modeling and Simulation

The Discrete Event System Specification (DEVS) [14] is a formalism which describes entities and behaviors of a system. It also allows the building of modular and hierarchical model compositions based on the closure-under coupling paradigm that means that the hierarchical models can be expressed to the single model. The DEVS formalism describes a system as a mathematical expression using set theory. It is a theoretically well-defined system formalism. The original DEVS formalism called the Classic DEVS had constraints that originated with the sequential operation of early computers and hindered the exploitation of parallelism, a critical element in modern computing. The parallel DEVS formalism equips bags to accommodate multiple input messages and the confluent function to handle simultaneous internal and external events.

There are two kinds of models in DEVS: atomic and coupled models. An atomic model depicts a system as a set of input/output events and internal states along with behavior functions regarding event consumption/production and internal state transitions. A coupled model consists of a set of atomic models, information of message connections between the atomic models, and input/output ports.

The Atomic model can be illustrated as a black box having a set of inputs(X) and a set of outputs(Y), or a white box specifying a set of states(S) with some operation functions (i.e., external transition function (δ_{ext}), internal transition function (δ_{int}), output function (λ), and time advance function (ta())) to describe the dynamic behaviors of the model.

The external transition function (δ_{ext}) carries the input messages and changes the system states. The internal transition function (δ_{int}) changes internal variables from the previous state to the next state when the time advance is expired and no events have occurred since the last transition. The output function (λ) generates an output event in the current state. The time advance (ta()) function determines the time to stay in the state after generating an output event. The atomic model is specified as follows:

$M = <X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta>$

where,

X: a set of inputs;

S: a set of states;

Y: a set of outputs;

δ_{int}: $S \rightarrow S$: internal transition function;

δ_{ext} : $Q \times X^b \rightarrow S$: external transition function;

λ: $S \rightarrow Y^b$: output Function;

ta : $S \rightarrow R^{+}_{0,\infty}$:time advance function.

X^b and Y^b are a set of bags over elements in X and Y.

$Q = \{(s,e) \mid s \in S, 0 \le e \le ta(s)\}$ is the set of total states where e is the elapsed time since last state transition.

A coupled model is the major class which embodies the hierarchical model composition constructs of the DEVS formalism [14]. A coupled model is made up of component models, and coupling relations that establish the desired communication links. A coupled model illustrates how to connect several component models together to form a new model. Two significant activities involved in coupled models specify its component models and define the couplings which create the desired communication networks. The coupled model is specified as follows:

$DN = < X, Y, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\} >$

where,

X: a set of external input events;

Y: a set of outputs;

D: a set of components names, for each i in D;

M_i: a component model;

$I_{i:}$ the set of influences for I; for each j in $I_{i;}$

$Z_{i,j:}$ the i-to-j output translation function.

A coupled model contains the following information:

- The set of components
- For each component, its influencees
- The set of input ports through which external events are received
- The set of output ports through which external events are sent
- The coupling specification consisting of:
    - The external input coupling (EIC) connects the input ports of the coupled to one or more of the input ports of the components
    - The external output coupling (EOC) connects the output ports of the components to one or more of the output ports of the coupled model

- Internal coupling (IC) connects output ports of components to input ports of other components

2.1.1. Discrete Event System Specification (DEVS) Modeling and Simulation

The DEVS modeling and simulation framework provides a very flexible and scalable modeling and simulation by separating models and simulators. The advantage of separation of modeling and simulation is an increase of adaptation of simulation in the various environments. For example, DEVS models can be simulated in distributed environment if simulators are altered to simulate models on the environment such as CORBA, HLA, and MPI [14].

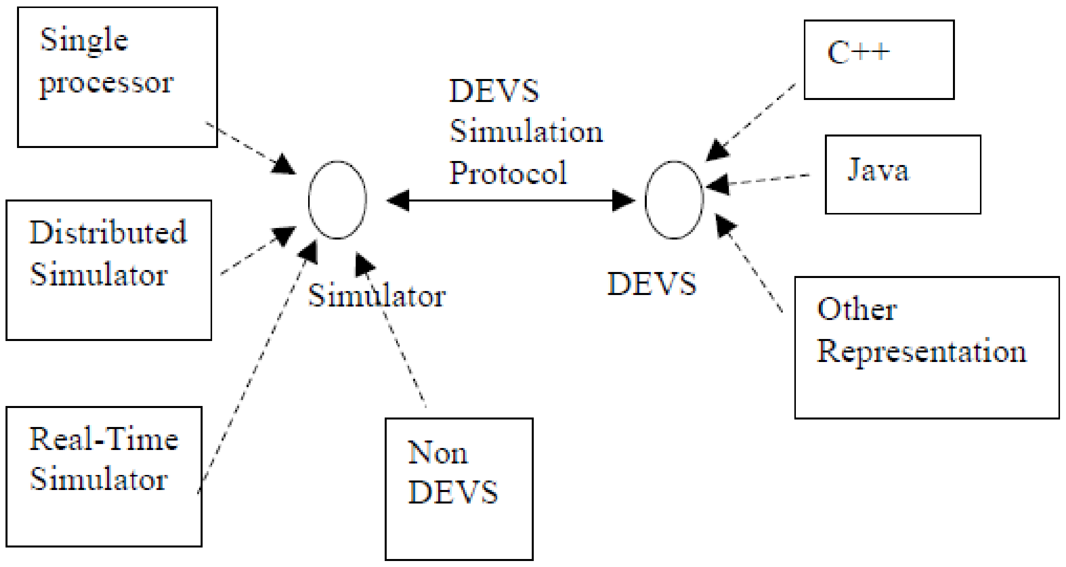


Figure 2-1 DEVS Modeling and Simulation Framework [15]

Figure 2-1 depicts DEVS modeling and simulation framework where simulator can be implemented in the single processor, distributed environment, real time manner, or non-

DEVS environment. The DEVS model can be implemented with C++, JAVA, or other implementation [27, 28, 75]. The simulator can simulate the DEVS models with simulation protocol. With this concept, the same models can be executed in different ways using different DEVS simulation protocols. Furthermore, middleware for parallel and distributed computing could be easily applied on separately developed DEVS models.

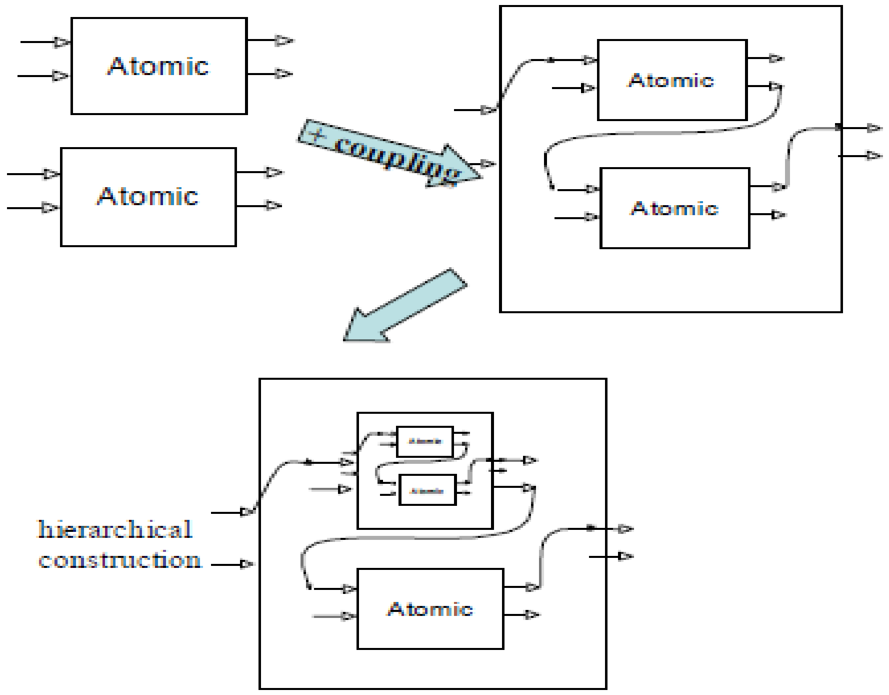


Figure 2-2 Coupled modules formed via coupling and their use as components [15]

Figure 2-2 represents the hierarchical model construction with components coupling. For example, a set of atomic models can be a coupled model by adding a coupling specification and the coupled model can be used as a component in a larger system. A hierarchical coupled model can be built by adding a set of model components as well as coupling information among these components.

The hierarchical construction and closure under coupling properties provide an excellent DEVS composition framework. Sometimes, the coupled model can not be used in the special circumstance such as middleware environment, so if the coupled model is considered as an atomic model and a simulator interprets the atomic model from the property of closure under coupling, the DEVS model will have more flexible simulation environment.

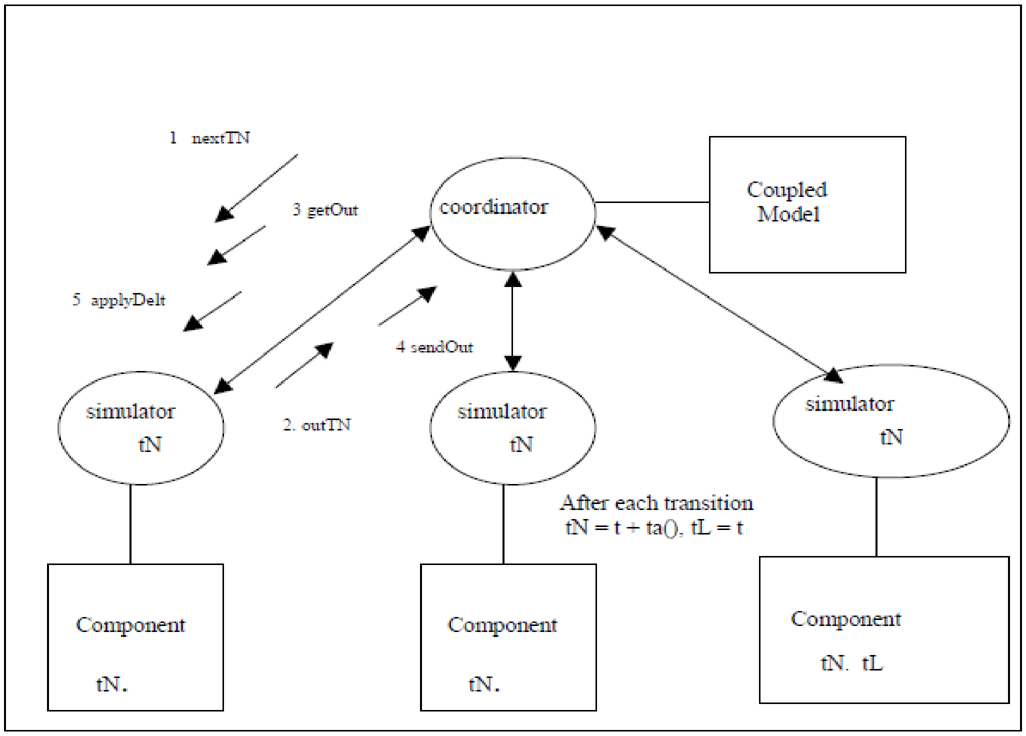


Figure 2-3 DEVS simulation protocol [15]

Figure 2-3 depicts the basic DEVS simulation protocol which is the key method to interconnect the modeling framework with simulation engines. There are two types of model handler called coordinator and simulator. Each handler manages a DEVS model, that is, a coordinator is assigned to the coupled model and simulators are assigned to the atomic models. The coordinator is responsible for overall simulation time management and execution. Simulation begins with the coordinator's sending *nextTN* to request *tN* from each of the simulators. All the simulators reply with their *tN*s in the *outTN* message to the coordinator. The coordinator sends to each simulator a *getOut* message containing the global *tN* selected from *tN*s as minimum *tN*. Each simulator checks if it is imminent (its *tN* = global *tN*) and if so, returns the output of its model in a message to the coordinator in a sendOut message. If it is imminent and its input message is empty, then it invokes its model's internal transition function. If it is imminent and its input message is not empty, it invokes its model's confluence transition function. If it is not imminent and its input message is not empty, it invokes its model's external transition function. If is not imminent and its input message is empty then nothing happens. The coordinator uses the coupling specification to distribute the outputs as accumulated messages back to the simulators in an *applyDelt* message to the simulators. For those simulators not receiving any input, the messages sent are empty.

The basic DEVS simulation protocol provides a key concept on how DEVS uses the simulators as well as how simulators interact with model components. In general, the DEVS based framework supports hierarchical, modular based modeling and simulation using reusable model components. The simulation protocol can be modified to increase simulation speed or to support real time simulation [30].

2.2. Service Oriented Architecture (SOA)

SOA [12] is a methodology with which a new application is created through integrating existing and independent business processes which are distributed over the networks. The business processes are called modules or services which communicate with each other, passing a message through the networks. This design concept requires interoperability between heterogeneous systems and languages and orchestration of services to meet the purpose of the creator.

2.2.1. Web Service

One of the implementations of the SOA concept is web service, which is a software system for communicating between a client and a server over a network with XML messages called Simple Object Access Protocol (SOAP) [15]. The web service makes the request of machine-to-machine or application-to-application communication possible with neutral message passing even though each machine or application is not in the same domain. Web services realize interoperability among different applications providing a standard means of communication and a platform independence.

The web services technologies architecture [13] is based on exchanging messages, describing web services, and publishing and discovering web service descriptions. The messages are exchanged by SOAP messages conveyed by internet protocols. Web services are described by Web Services Description Language (WSDL) [14] which is a XML based language providing required information such as message types, signatures of operations, and a location of a service, for clients to consume the services. Publishing and discovering WSDLs is managed by Universal Description Discover and Integration (UDDI), which is a platform-independent and XML style registry. In other words, three roles are classified in the architecture: a service provider, a service discovery agency

(UDDI), and a service requestor. The interaction of the roles involves publishing, finding, and binding operations. A service provider defines a service description for a web service and publishes it to a service discovery agency. This operation publishes operations between the service provider and the service discovery agency. A service requestor uses a finding operation to retrieve a service description locally or from a discovery agency and uses the service description to bind it with a service provider and invoke or interact with the web service implementation. Figure 2-4 illustrates the basic Web services architecture describing three roles and operations with WSDL and SOAP.

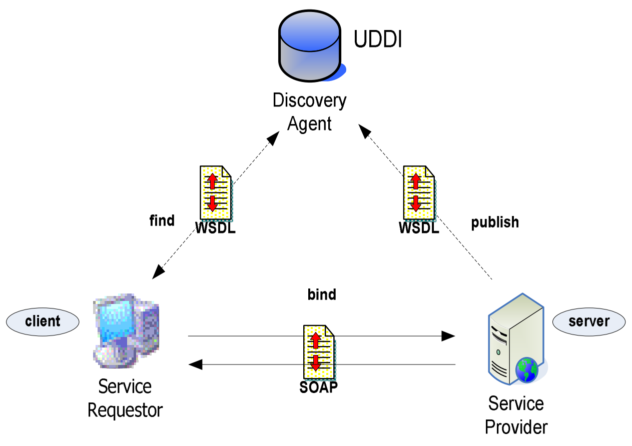


Figure 2-4 Web Services Architecture

Whereas a web service is an interface described by a service description, its implementation is the service which is a software module provided by the service provider (server) on the network accessible environment. It is invoked by or interacts with a service requestor (client).

Web services are invoked in many ways, but the common use of web services is categorized to three methods such as Remote Procedure Call (RPC) [16], Service Oriented Architecture (SOA) [16], and Representational State Transfer (REST) [16]. RPC

Web services was the first web services approach which had a distributed function call interface described in the WSDL operation. Though it is widely used and upheld, it does not support the loosely coupled concept for reasons of mapping services directly to language-specific functions calls. Another web service is an implementation of SOA concepts, which means a message is an important unit of communication regarded as “message-oriented” services. This approach supports a loose coupling concept focusing on the contents of WSDL. REST Web services which focus on the existence of resources rather than messages or operations. It considers WSDL as a description of SOAP messaging over HTTP, or is implemented as an abstraction on top of SOAP.

2.2.2. Simple Object Access Protocol (SOAP)

SOAP [17] is a simple and lightweight XML-based mechanism for creating structured data packages that can be exchanged between network applications. SOAP consists of four fundamental components: an envelope that defines a framework for describing message structure, a set of encoding rules for expressing instances of application-defined data types, a convention for representing remote procedure calls (RPC) and responses, and a set of rules for using SOAP with HTTP. SOAP can be used in combination with a variety of network protocols such as HTTP, SMTP, FTP, RMI/IIOP, or a proprietary messaging protocol [62, 63].

SOAP provides a way to communicate between applications running on different operating systems, and a SOAP message is an ordinary XML document containing the following elements as seen in figure 2-5:

- An Envelope element that identifies the XML document as a SOAP message
- A header element that contains header information
- A body element that contains call and response information
- A fault element containing errors and status information

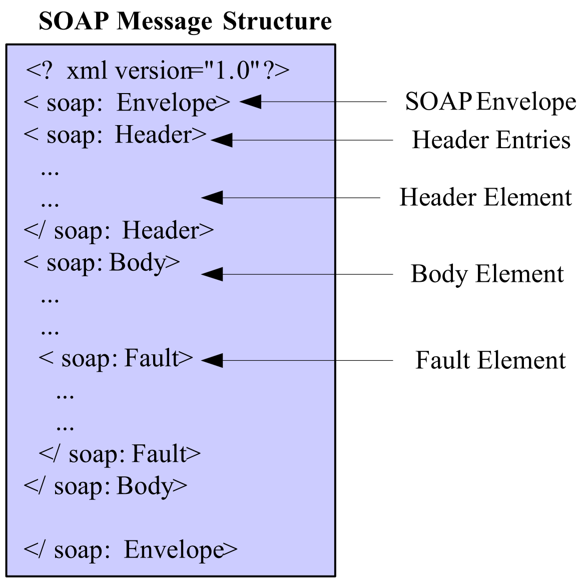


Figure 2-5 The structure of SOAP

The required SOAP envelop element is the root element of a SOAP message. The namespace defines the envelope as a SOAP envelope, and if a different namespace is used, the application generates an error and discards the message. The *encodingStyle* attribute is used to define the data types used in the document. This attribute may appear on any SOAP element, and it will apply to that element's contents and all children.

The optional SOAP header element contains application specific information like authentication, and payment. SOAP defines three attributes in the default namespace. These attributes are *mustUnderstand*, actor, and *encodingStyle*. The attributes defined in the SOAP header define how a recipient should process the SOAP message. The actor attribute is used to address the header element to a specific endpoint. The *encodingStyle* attribute is used to define the data types used in the document. The required SOAP body element contains the actual SOAP message intended for the ultimate endpoint of the message. The optional SOAP fault element is used to indicate error messages.

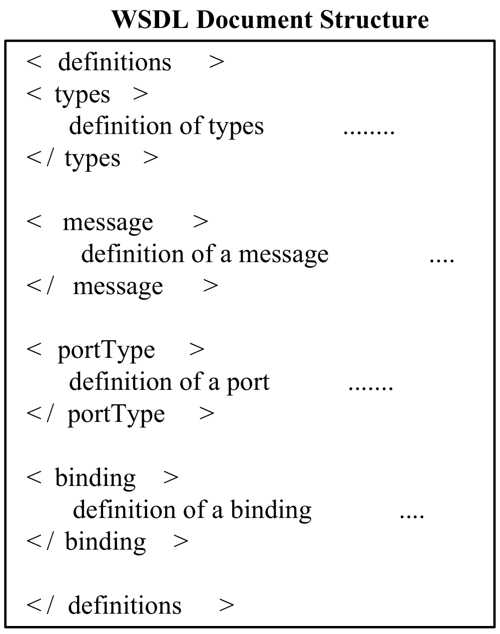
SOAP is currently the standard for XML messaging for a number of reasons. First, SOAP is relatively simple, defining a thin layer that builds on top of existing network technologies such as HTTP that are already broadly implemented. Second, SOAP is flexible and extensible in that rather than trying to solve all of the various issues developers may face when constructing Web services, it provides an extensible, composable framework that allows solutions to be incrementally applied as needed. Thirdly, SOAP is based on XML. Finally, SOAP enjoys broad industry and developer community support.

The following details explain more about SOAP [18-19]

- Specification: SOAP is not a product but a document that describes the characteristics of a piece of software.
- Ubiquitous application: SOAP is a high level of abstraction that any operation system and programming language combination could be used to create SOAP-compliant programs.
- XML-Basis: SOAP is designed on top of XML, which means that SOAP documents are XML documents constructed to a tighter set of specifications.

2.2.3. Web Services Description Language (WSDL)

WSDL is a document written in an XML format published for describing Web services. It specifies the location of the service and the operations which the service exposes. WSDL describes how to communicate using the web service; namely, the protocol bindings and message formats required to interact with the web services listed in its directory. The supported operations and messages are described abstractly, and then bound to a concrete network protocol and message format.



WSDL Document Structure

```
< definitions   >
< types  >
    definition of types        ........
</ types  >

<  message    >
     definition of a message          ....
</  message    >

< portType   >
    definition of a port        .......
</ portType   >

< binding   >
    definition of a binding         ....
</ binding   >

</ definitions   >
```

Figure 2-6 WSDL document structure

The WSDL document structure consists of *portType, message, types,* and *binding* as seen in figure 2-6. The *portType* element describes a web service, the operations that can be performed, and the messages that are involved. It can be compared to a function library in a traditional programming language. The *message* element defines the data elements of an operation. The *types* element defines the data type that are used by the web service. The *binding* element defines the message format and protocol details for each port.

WSDL is often used in combination with SOAP and XML Schema to provide web services over the internet. A client program connecting to a web service can read the WSDL to determine what functions are available on the server. Any special data types used are embedded in the WSDL document in the form of XML Schema. The client can use SOAP to actually call one of the functions listed in the WSDL.

2.3. Apache AXIS2

Apache AXIS2 [20] is the core engine for web services, supports SOAP 1.1 and SOAP 1.2, and has integrated support for the widely popular REST style of web services. It gives both a WS- style interface and REST/POX style interface to the same web service implementation simultaneously. Apache AXIS2 is a SOAP engine which processes the SOAP message in and out services.

Figure 2-7 represents the AXIS2 architecture diagram which consists of core components and other components. The core components are XML processing model (AXIOM), SOAP processing model (handler framework), and information processing model (contexts and descriptions). Other components include deployment model, transports, client API, and code generation model. AXIOM (AXIs2 Object Model) is the base for AXIS2, where any incoming SOAP message is represented as AXIOM inside AXIS2. It is based on a pull parser technique in which the invoker has the full control on the parser.

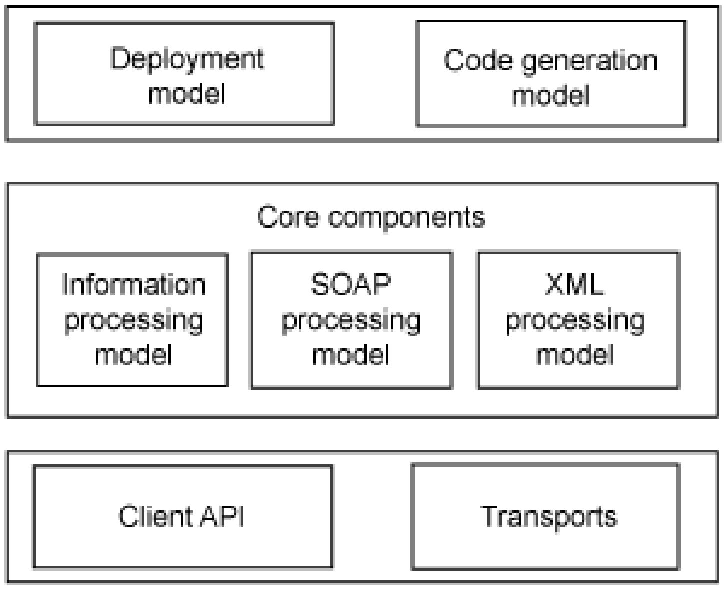


Figure 2-7 AXIS2 architecture diagram [20]

The handler framework has a special handler called a receiver which receives messages and is used to call the provider component, a sender which sends messages and invokes the outflow handler chain, and a dispatcher which finds the service. The handlers are the execution units and phases are logical handler collections. The deployment provides an AXIS archive called *.aar* file which is like a jar file with all the service classes and the service description. It can be uploaded through the web or directly through the file system. The deployment model has hot deployment and hot update functions with which it can deploy and update services without shutting down the system. The client API provides facility for synchronous/asynchronous invocations whose supported styles are in-out sync, in-out async, and in-only. The code generation model provides WSDL2Java or Java2WSDL tools to make code generation easy.

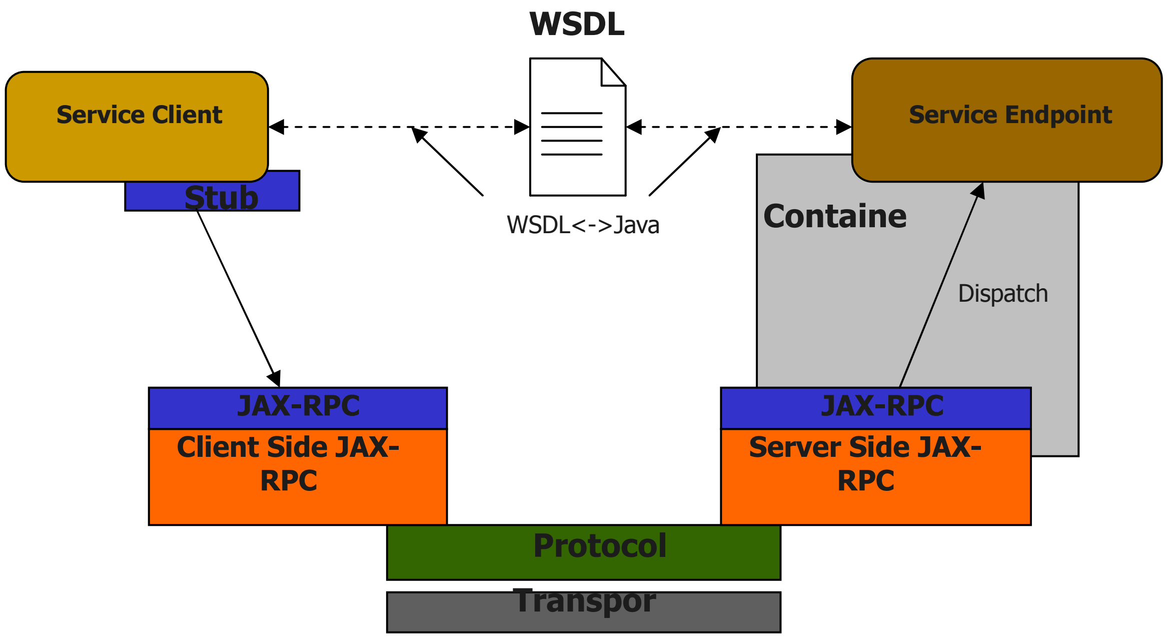


Figure 2-8 represents JAX-RPC physical architecture for web service invocation. The service client invokes the service through a stub class which is implemented with client side JAX-RPC. An invoking message reaches to server side JAX-RPC which dispatches the message to the service endpoint.

AXIS2 provides two methods for creating web services. They are top-down method with WSDL, and bottom-up method with codes. The top-down method uses *WSDL2Java* utility to generate server side codes. We add some codes to the server side codes to implement each service. After service codes are complete, we create an *.aar* file containing all resources and deploy the *.aar* file to a server. The bottom-up method with codes begins with creating service codes, creates a service descriptor, an *.aar* file including all resources, and deploys the *.aar* to a server. To consume web service, we need to create the client stub using *WSDL2Java* utility and create the client application using the generated stub to call a service.

2.4. Interoperability studies

Interoperability is required in the integrated system with complex and distributed software modules to create a new system. It is not easy for software modules in the different domains to interoperate with other software modules. Sometimes we consider the interoperable problems among heterogeneous systems as message mapping problems. It is partly true, but the message mapping is not always true to create interoperable systems. At this point, we have a question regarding what is the definition of interoperability. The IEEE has four definitions of interoperability:

- The ability of two or more systems or elements to exchange information and to use the information that has been exchanged.
- The capability for units of equipment to work together to do useful functions
- The capability, promoted but not guaranteed by joint conformance with a given set of standards, that enables heterogeneous equipment, generally built by various vendors, to work together in a network environment.
- The ability of two or more systems or components to exchange information in a heterogeneous network and use the information.

We can find more definition of interoperability in the DoD:

- The ability of systems, units, or forces to provide services to and accept services from other systems, units, or forces, and to use the services so exchanged to enable them to operate effectively together [21].

- The condition achieved among communications-electronics systems or items of communications-electronics systems equipment when information or services can be exchanged directly and satisfactorily between them and/or their users. The degree of interoperability should be defined when referring to specific cases. For the purposes of this instruction, the degree of interoperability will be determined by the accomplishment of the proposed Information Exchange Requirement (IER) fields [22].
- (a) Ability of information systems to communicate with each other and exchange information. (b) Conditions, achieved in varying levels, when information systems and/or their components can exchange information directly and satisfactorily among them. (c) The ability to operate software and exchange information in a heterogeneous network (i.e., one large network made up of several different local area networks). (d) Systems or programs capable of exchanging information and operating together effectively [23].

From the above definitions, we can partly understand the meaning of interoperability. According to the complexity of interoperability used, terms to define the interoperability are changed. For example, interoperability is satisfied if some systems have capability for communication and exchange of information. But in some situations, interoperability conditions could be different.

Levels of information system interoperability (LISI) are categorized into five levels according to the increasing levels of complexity of systems interoperability. The five levels are defined in the table 2-1.

Table 2-1 Five levels in LISI

Level 0	Isolated interoperability in a manual environment between stand-alone system
Level 1	Connected interoperability in a peer-to-peer environment
Level 2	Functional interoperability in a distributed environment
Level 3	Domain based interoperability in an integrated environment
Level 4	Enterprise-based interoperability in a universal environment

LISI concentrates on technical interoperability and the complexity of interoperations between systems and does not mention the environmental and organizational issues that affect the construction and maintenance of interoperable system.

The Organizational Interoperability Maturity Model (OIM) broadens the LISI model into the more abstract layers of command and control support [72]. OIM in table 2-2 focuses on the human-activity and user aspects of military operations.

Table 2-2 Level of OIM

Level 0	Independent
Level 1	Ad hoc
Level 2	Collaborative
Level 3	Integrated (combined)
Level 4	Unified

NATO C3 Technical Architecture (NC3TA) reference model for interoperability provides four degrees of interoperability in table 2-3.

Table 2-3 Degree of NC3TA

Degree 1	Unstructured Data Exchange: exchange of human-interpretable unstructured data such as text
Degree 2	Structured Data Exchange: exchange of human-interpretable structured data intended for manual and/or automated handling.

Degree 3	Seamless sharing of Data: automated sharing of data amongst systems based on a common exchange model.
Degree 4	Seamless Sharing of Information: universal interpretation of information through data processing based on cooperating applications.

NC3TA categorized how operational effectiveness could be enhanced by structuring and automating the exchange and interpretation of data and was updated to closely reflect the LISI model.

Levels of Conceptual Interoperability (LCIM) model addresses levels of conceptual interoperability that go beyond technical models like LISI [4]. This model is intended to a link between conceptual design and technical design.

Table 2-4 Level of LCIM

Level 0	System specific data: black box components with no interoperability or shared data
Level 1	Documented data: shared protocols between systems with data accessible via interfaces.
Level 2	Aligned static data: common reference model with the meaning of data unambiguously described. Systems are black boxes with standard interfaces. However, even with a common reference model, the same data can be interpreted differently in different systems.
Level 3	Aligned dynamic data: Use of data is defined using software engineering methods like Unified Modeling Language.
Level 4	Harmonized data: Non-obvious semantic connections are made apparent via a documented conceptual model underlying components.

Three linguistic levels of interoperability have been defined [8, 25]. These levels are illustrated in table 2-5:

Table 2-5 Linguistic levels of interoperability

Linguistic Level	**A collaboration of systems or services interoperates at this level if:**

Pragmatic – how information in messages is used	The receiver reacts to the message in a manner that the sender intends (assuming non-hostility in the collaboration).
Semantic – shared understanding of meaning of messages	The receiver assigns the same meaning as the sender did to the message.
Syntactic – common rules governing composition and transmitting of messages	The consumer is able to receive and parse the sender's message

The linguistic levels of interoperability focus on the meaning of messages interpreted in the view of syntactic, semantic, and pragmatic, whereas, LISI categorizes the complex system into five levels for interoperability. Implementations of the Interoperability system vary according to the domains and requirements. The above mentioned research describes a generic approach of interoperability in the specific domains such as DoD and SoS.

CHAPTER 3. OVERALL ARCHITECTURE OF DEVS SIMULATOR SERVICES INTEROPERABILITY

The overall architecture of DEVS simulator services interoperability consists of web technology and namespace concepts. The web service provides common infrastructure of system/language interoperability and the namespace presents a look-up table for messages which are passed between services.

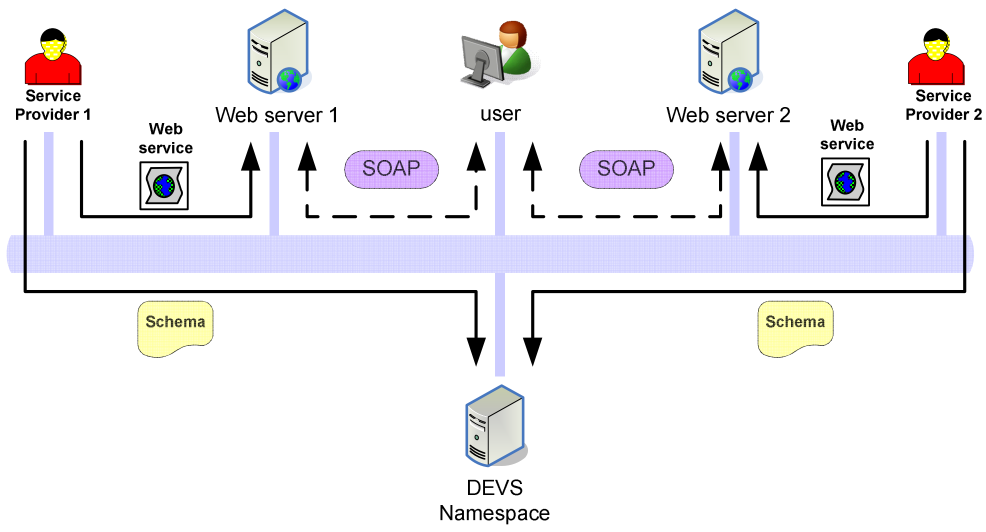


Figure 3-1 Creation and consuming of DEVS simulator services

Figure 3-1 explains the roles of service providers, a user, and a DEVS namespace to illustrate how DEVS simulators interoperability works. A service provider generates web services with a specific language, which contain DEVS models and uses predefined service operations, and before loading the web services, the provider registers message types used in the DEVS model to the DEVS namespace. Other provider gets the schema containing a message type from the DEVS namespace to create a web service which is interoperable with web services with the same message types. The web service contains a location of the DEVS namespace and information of types of messages.

The user integrates DEVS simulator services whose information is displayed as a XML formed document conforming to its schema document called devswsintegrator.xsd and executes the integration of DEVS simulator services through parsing the XML document.

In this chapter, we discuss a structure and a design of DEVS simulator service conforming to DEVS simulation protocol and the DEVS namespace containing data types for DEVS models in the DEVS simulator services. Also, we mention a structure and a function of a document of devswsintegrator.xsd, and an extraction of a XML document resulting from the user's integration of DEVS simulator services.

3.1. System of Interoperability of DEVS Simulator Services

The interoperability system of DEVS simulator services consists of three parts: a DEVS namespace, DEVS simulator services, and DEVS simulator service integration and execution (DSSIE). The DEVS namespace is a schema that contains message type definitions. It is used to recognize message types between distributed or different systems when the systems need to cooperate in a system of systems [26]. The message types of each service are registered in the DEVS namespace before the service publishes in the server.

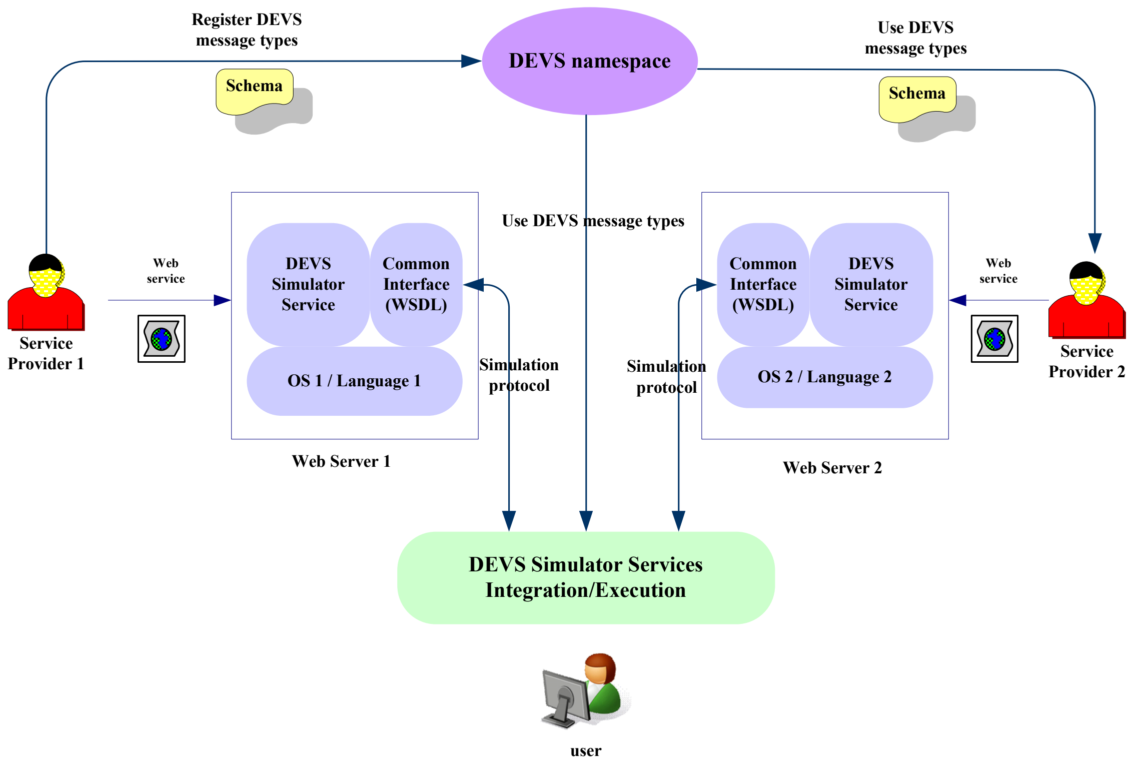


Figure 3-2 Overall system of DEVS simulator services interoperability

The DEVS simulator service has a common interface to provide interoperability between different platforms or different languages. The common interface is called WSDL defining operations, message types, and the location of the service. To generate a common interface for the DEVS simulator service, there are different ways according to implementation of web service referred to web service middleware. Through the middlewares, the DEVS simulator service can be implemented on various operating systems and computer languages. Through various implementations of the DEVS simulator service, SOAP messages, which are used as request and response messages of operations of the web service, provide loosely coupling and neutral message passing.

Each middleware for the web service provides functions to convert SOAP messages to an instance of a specific language and vice versa.

In figure 3-2, two DEVS simulator services provide common interfaces on the different platforms. A common interface contains operations for DEVS simulation protocol to simulate DEVS models in different services.

DSSIE has two functions, the integration of the DEVS simulator services based on message types and the execution of the integrated system. The integration of the DEVS simulator services is performed by a GUI called a DEVS simulation service integrator which uses the DEVS namespace to verify if couplings between two services are possible or not. The data on the integrator are written to a XML document sent to the executor which simulates DEVS simulator services. The executor adopts Java Architecture for XML Binding (JAXB) API to make handling the XML easy. In the figure 3-2, the DSSIE obtains DEVS message types of DEVS simulator services from the DEVS namespace to integrate services and simulate the DEVS services with simulation protocols.

3.2. The DEVS namespace

The DEVS namespace is storage for types of messages which are used in DEVS models. The types are expressed into an element of XML schema that describes a structure of the XML document. XML schema assigns a unique name to each element. For example, if the name of the element is Job, Job element is unique in the schema document. Uniqueness of a type gives clearness for message passing between systems on interoperable operation.

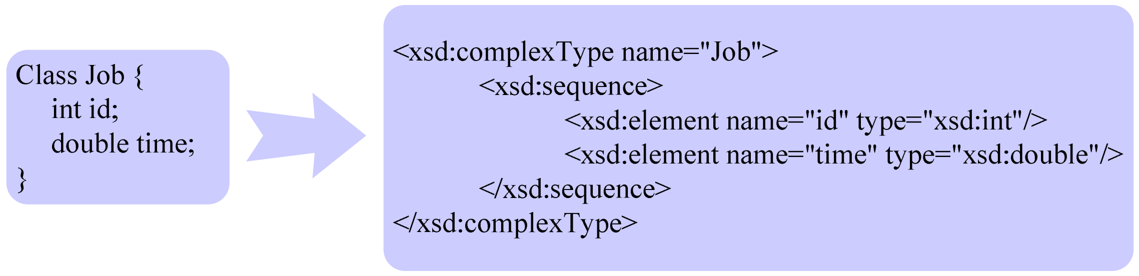
WSDL for a DEVS simulator service defines data types for each operation. When the web service communicates with a user, the operations of the web service receive an argument as XML document embraced in a SOAP message. The XML document is created in conformance with a type of schema in WSDL. The return value of operations is generated above the procedure. The data types in WSDL are just defined for operations of a DEVS simulator not a DEVS model. In the view of simulation, the structure of a DEVS message consists of a set of content which has a port name and an object. The DEVS model uses an object as a message. That means the message type has no common type covering all DEVS messages in the different languages. To overcome this problem, a DEVS message is converted to a XML document in the web service level. This approach works if DEVS simulator services use the same messages in the DEVS models.

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsd:schema xmlns:ns0="http://devs.service"  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" targetNamespace="http://devs.service">
<xsd:element name="EFP">
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element name="Job" type="ns0:Job"/>
                </xsd:sequence>
        </xsd:complexType>
</xsd:element>
<xsd:complexType name="Job">
        <xsd:sequence>
                <xsd:element name="id" type="xsd:int"/>
                <xsd:element name="time" type="xsd:double"/>
        </xsd:sequence>
</xsd:complexType>
```

Figure 3-3 The *DEVSNamespace.xsd*

To integrate DEVS simulator services in different platforms or languages, information of model level messages should be known to a user. To meet this end, we employ a DEVS namespace to the system for the interoperability of DEVS simulator services. The DEVS namespace is a document called *DEVSNamespace.xsd* that we can access through the network.

Figure 3-3 shows the *DEVSNamespace.xsd* document which has a data type named as *Job*. In the document, *xsd* is a prefix referring to “http://www.w3.org/2001/XMLSchema” site containing primitive type definition. For example, *type* = *xsd:int* means *type* is *int* value defined in the “http://www.w3.org/2001/XMLSchema”. [67] has more information about the meaning of basic elements of XML schema and how a schema document is composed.



```
Class Job {
    int id;
    double time;
}
```

```
<xsd:complexType name="Job">
        <xsd:sequence>
                <xsd:element name="id" type="xsd:int"/>
                <xsd:element name="time" type="xsd:double"/>
        </xsd:sequence>
</xsd:complexType>
```

Figure 3-4 Conversion of *Job* class to schema data type

Figure 3-4 shows the conversion of a language class to a schema type. If a *Job* class is used in the DEVS model, the *Job* class should be expressed as a corresponding schema data type. In the example, *Job* class has two variables named *id* and time which are assigned to *int* and double type, respectively. The schema data type represents all variables in the class. The name of class is the name of a data type and variables become sub elements of the data type. The sub elements are assigned to primitive data types like variables in the class.

Conversion of a class to a schema is performed by a service provider. The schema document resulting from the conversion is registered into the DEVS namespace. Figure 3-5 depicts a procedure of registration of a schema data type into a DEVS namespace server. The procedure starts with sending a schema document to a web service which has four operations. One operation, called *checkSchema*, has one argument for the schema document and a Boolean return type to send a result of checking if the schema type is in the DEVS namespace. Another operation, called *registerSchema*, is for registering the schema document to the DEVS namespace. The *getDomains* and *getMessageTypes* operations are used to search the DEVS namespace and get a specific message type.

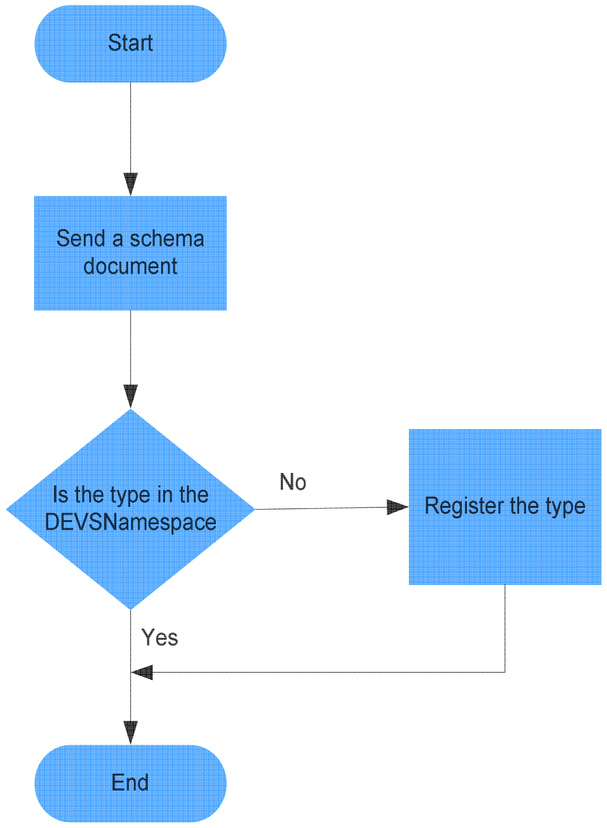


Figure 3-5 The registration of a schema document

3.3. The Structure and design of DEVS Simulator Service

3.3.1. The structure of DEVS simulator service

To implement a web service for a DEVS simulator, a middleware for helping create the web service is needed such as Apache eXtensible Interaction System (AXIS2) [9] or .Net framework [12]. The middleware provide API to make building a web service easy, hiding complicating network programming. AXIS2 can be adopted as web service middleware which is embodied by Java program while .Net can be used for C++ or C# user.

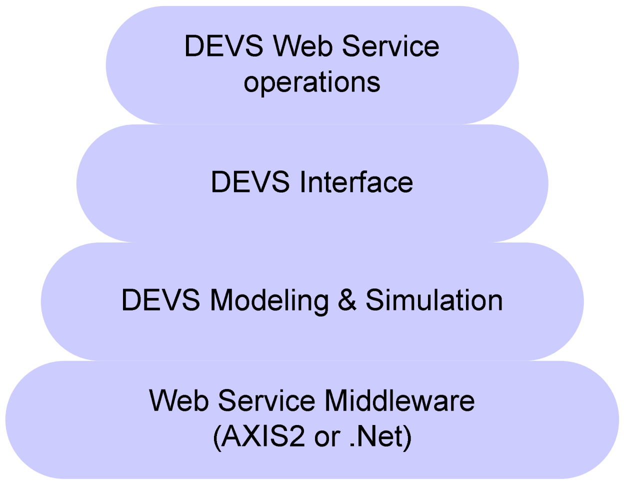

DEVS Web Service operations
DEVS Interface
DEVS Modeling & Simulation
Web Service Middleware (AXIS2 or .Net)

Figure 3-6 The software stacks of a DEVS simulator service

Figure 3-6 shows that DEVS simulator service is supported by several APIs to implement a DEVS simulator services interoperability system. The bottom layer is a web service middleware which provides the network connection environment and a handler of SOAP messages between a web service and a client program. The handler of SOAP messages includes a convertor of SOAP messages to instances of application and vice versa. This layer can be selected according to an operation system and a language that supports the web service.

DEVS modeling and simulation (DEVS M&S) API enable DEVS M&S to be used in the web service. DEVS M&S can be implemented with different environments and languages. For example, if a service provider uses a Java language, DEVS M&S for Java should be used to generate a DEVS simulator service. What language is used in the middleware decides what kind of implementation of DEVS M&S is used. DEVS M&S is embodied with Java, C++, C#, and so on [27, 28].

The role of DEVS interface is connection between web service operations which is described by WSDL, and DEVS M&S. The DEVS interface is required because DEVS M&S API do not support what web service operations want. It helps web service operations extract information in the DEVS M&S.

Web service operations are described in the WSDL and in a class which has methods whose name is the same as the web service operations. The web service operations are designed to reflect the provider's intention. The operations of DEVS simulator service is selected to conform to DEVS simulation protocol.

3.3.2. Design of the DEVS simulator service

The design of the DEVS simulator service starts from consideration of what is the role of the DEVS simulator service. First of all, the DEVS simulator service is capable of handing the information of a DEVS model to a requestor in order to make the DEVS model simulated with other DEVS simulator services. Second, the DEVS simulator service passes the information of schema location and message types to a client to let the client know information of schema location and message types of the DEVS model. Last, the user should know the result of simulation after finishing the execution of the integration of DEVS simulator services. Therefore, reporting functions are included in the design of the DEVS simulator service.

As a result of all considerations, DEVS simulator service has three categories: DEVS simulation protocol operations, schema location and message type operations, and reporting function operations. Figure 3-7 represents three categories of operations and signatures of operations.

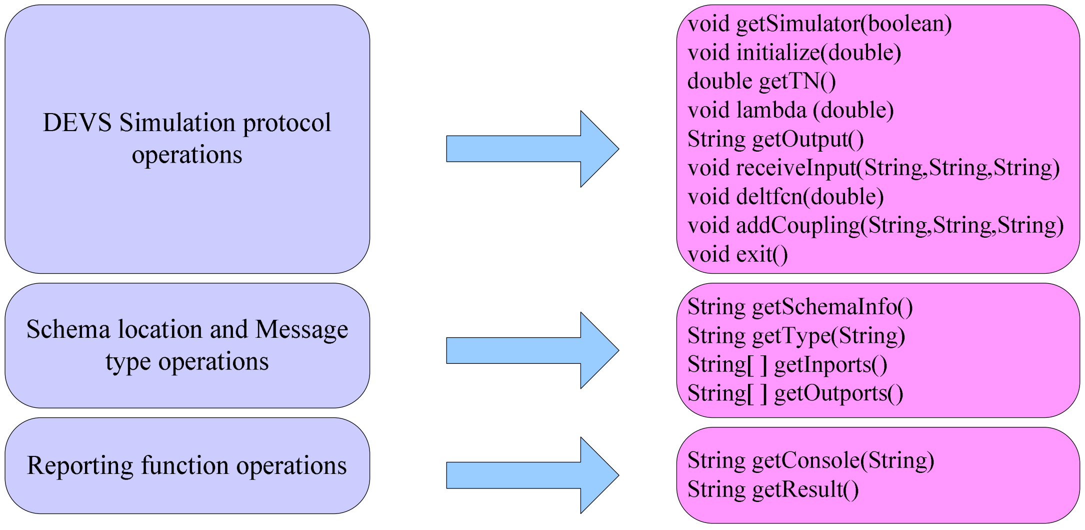

DEVS Simulation protocol operations
Schema location and Message type operations
Reporting function operations
void getSimulator(boolean)
void initialize(double)
double getTN()
void lambda (double)
String getOutput()
void receiveInput(String,String,String)
void deltfcn(double)
void addCoupling(String,String,String)
void exit()
String getSchemaInfo()
String getType(String)
String[ ] getInports()
String[ ] getOutports()
String getConsole(String)
String getResult()

Figure 3-7 The operations of DEVS simulator service

The operations for DEVS simulation protocol at the top of the figure 3-7 are utilized when DEVS simulation is executed by a user. There are nine operations: *getSimulator, initialize, getTN, lambda, getOutput, receiveInput, deltfcn, addCoupling,* and *exit*. The *getSimulator* operation decides which simulator is used. There are two kinds of simulators which are for centralization and decentralization. If its argument is set to *false*, the DEVS simulator service uses a simulator for centralization. If *true*, it uses a simulator for decentralization. The *addCoupling* operation is used in case of a simulator for decentralization to let the simulator know coupling information for sending messages to a destination service. When a simulator is selected, the simulator has a DEVS model.

DEVS simulation protocol starts with the initializing operation which is called when the simulation begins. The *getTN* operation returns next internal event time (*TN*) to a coordinator which is in the DEVS simulator services integration and execution as seen in figure 3-2. The lambda operation generates output messages if the model has an internal event. The *getOutput* operation returns output messages which consist of the XML document to the coordinator which looks up the coupling table and requests the invocation of the *receiveInput* operation to a corresponding DEVS simulator service. The *receiveInput* operation sends output messages, input port name, and output port name to the target service. The input port name is used to generate DEVS messages in the target service. Thereafter, the *deltfcn* operation changing the state of the model and scheduling *TN* is called to all DEVS simulator services. This is one cycle of DEVS simulation protocol. The simulation protocol is repeated until meeting the certain condition to stop the simulation such as infinity of *TN* of all simulator services, and the number of simulation protocol cycles.

The operations for schema location and message type in middle of the figure 3-7 have four operations which are *getSchemaInfo, getType, getInports,* and *getOutports*. Each simulator service has information of schema location and model’s message types which is registered in the schema repository called DEVS namespace and exposes the location of the schema, the names of input ports and output ports, and message types used in the input or output ports with the four operations. The *getSchemaInfo* returns the location of schema, the *getType* returns the type for an input or output port when sending a port name, the *getInports* returns an array of names of input ports of the model, and the *getOutports* returns an array of names of output ports of the model. These operations are used when DEVS simulator services are integrated based on matching message types between the models.

The operations of the reporting function in the bottom of the figure 3-7 has two operations, that is, *getConsole* and *getResult*. The *getConsole* operation returns a document produced by the simulator service during simulation protocol cycles. The document can be used to check any bug in the model and validate if the model in the simulator service is appropriately working. The *getResult* operation returns the result of the simulation if the simulator service generates data written in the result document located in the specific place.

3.4. WSDL of the DEVS Simulator Service

Based on the operations as seen in the figure 3-7, we can obtain WSDL for the DEVS simulator service by using a tool or creating a web service. AXIS2 provides a tool named *Java2WSDL* to create WSDL conforming to the Java class. The procedure of creating WSDL of the DEVS simulator service is as follows.

- Make a Java class including all operations

- Compile the class
- Apply a JAVA2WSDL tool with options

Figure 3-8 represents the Java interface for the DEVS simulator service to generate WSDL. We add two operations for the future named *isReady4delta* and *simulateReal*. Once creating the Java interface, we should compile it for a Java2WSDL tool.

```
public interface Simulator {
   public void getSimulator(boolean isRT);
   public void addCoupling(String portFrom, String portTo, String ipServiceTo);
   public String getConsole(String clientIp);
   public void exit();
   public void initialize(double t);
   public double getTN();
   public void lambda(double t);
   public String getOutput();
   public void receiveInput(String portFrom, String msg, String portTo);
   public void deltfcn(double t);
   public String getResult();
   public String getSchemaInfo();
   public String getType(String port);
   public String[] getInports();
   public String[] getOutports();
   public boolean isReady4delta();
   public void simulateReal();
}
```

Figure 3-8 Java Interface of Simulator

Figure 3-9 shows how to use the Java2WSDL tool. The tool requires as options the name of WSDL document, the location of the DEVS simulator service, and the target names-pace. After applying the Java2WSDL, we get the WSDL for the DEVS simulator service.

The other way to get WSDL for the DEVS simulator service is from the web service. When completing uploading the web service to the web server, the web service middleware creates the WSDL for the web service. The WSDL can be seen through a web browser if the web service is working properly.

```
Java2WSDL -o Simulator.wsdl
    -l"http://localhost:8080/axis/services/Simulator"
    -n  "urn:http://devs.service" -p"service.devs" "urn:http://devs.service"
    service.devs.Simulator
where :
-o indicates the name of the output WSDL file
-l indicates the location of the service
-n is the target namespace of the WSDL file
-p indicates a mapping from the package to a namespace. There may be multiple mappings.
the class specified contains the interface of the webservice.
```

Figure 3-9 The usage of the Java2WSDL tool

3.5. Creation of the DEVS Simulator Service

There are two ways to create a web service. One way is to use the WSDL created by using a Java2WSDL tool. The other way is to directly write the codes for the web service based on defined operations in 3.3.2. When using WSDL, we can use command line or the plug-in for the IDE such as Eclipse or intelliJ IDEA [73].

Figure 3-10 displays the procedure to generate a web service. In the left hand side, we can create the web service with the WSDL where a *wsdl2java* tool automatically creates a skeleton Java file, message handling file, and service.xml. The service provider adds contents to the operations from the skeleton document. In this procedure, supporting classes can be used in the skeleton document. For example, in case of a DEVS simulator service, DEVS M&S API and DEVS interface API can be used. In the *service.xml*, the information of the web service is written to let the web service middleware know what the class of the web service is and which message handlers should be used for operations. Thereafter, all codes should be compiled and archived to be easy to deploy the web service.

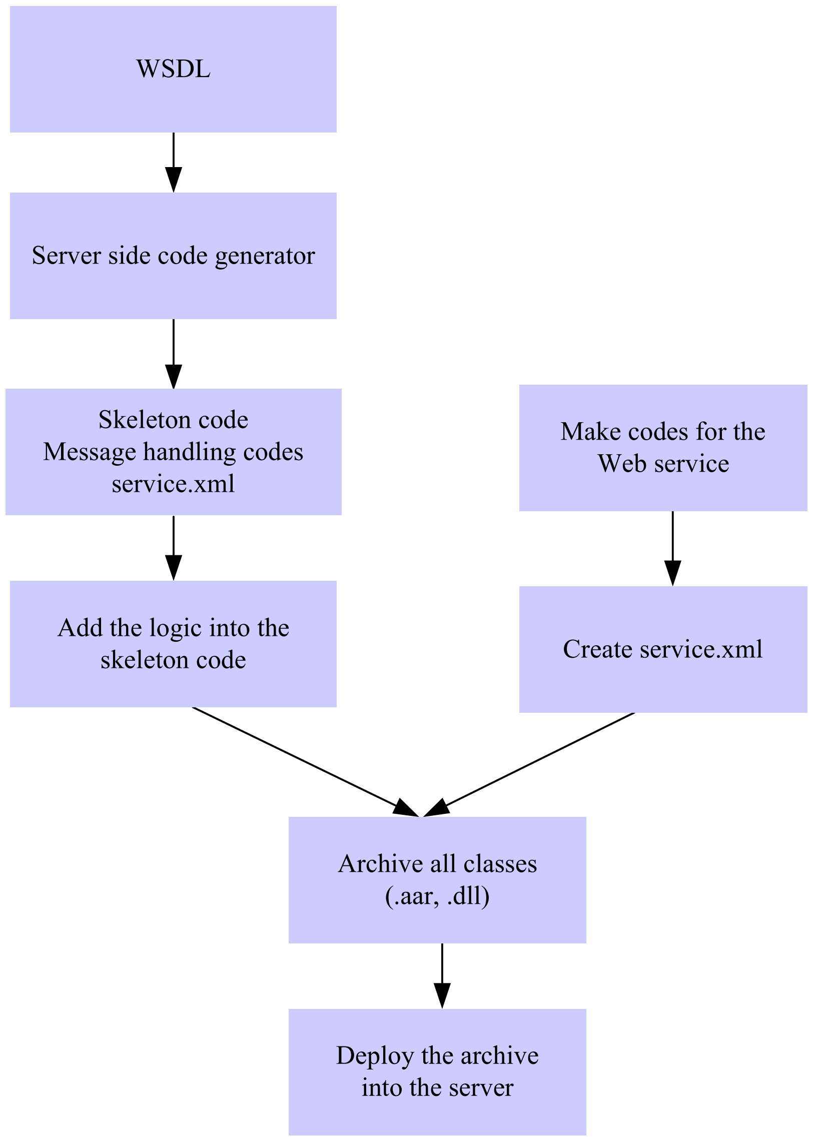

WSDL
Server side code generator
Skeleton code
Message handling codes
service.xml
Add the logic into the skeleton code
Make codes for the Web service
Create service.xml
Archive all classes (.aar, .dll)
Deploy the archive into the server

Figure 3-10 The procedure of creation of the web service

On the right hand side, we can create a web service directly from the source codes without WSDL. The main source code should contain all operations defined in 3.3.2. After adding all logics in the operations, next procedure is same as that of WSDL based web service generation.

3.6. DEVS Simulator Service Integration and Execution

In this section, we discuss the invocation of a web service and composition of interoperable DEVS simulator services with message matching coupling. We show how to execute the integrated DEVS simulator services. The integration of DEVS simulator services is performed through the GUI with which we can easily get a XML document describing information of DEVS simulator services and of coupling between the services. Also, the XML document is used as an input during the execution of Interoperable DEVS simulator services.

3.6.1. Invocation of a DEVS simulator service

There are two approaches to make a client program which invokes the web service. One is to use the WSDL with which a client codes are generated using a client code generation tool. The other is to use middleware API for dynamic invocation of web services. In figure 3-11, two kinds of invocations begin with handling the WSDL. The left side in the figure 3-11 depicts how an application uses a tool based generated code called a client stub [20]. The client stub has network connection information, signatures of operations, SOAP message handler, and XML to class converter vice versa. A user can access to a web service with its client stub. There are two types of invocation of a web service, that is, synchronous and asynchronous invocations. When using the tool of AXIS2, a *callbackhandler* code is generated to support asynchronous invocation of the web service [20]. The asynchronous invocation can reduce total time of invocations when lots of web services are invoked. If the *callbackhandler* is not used in the application, the invocation of the web service is synchronous.

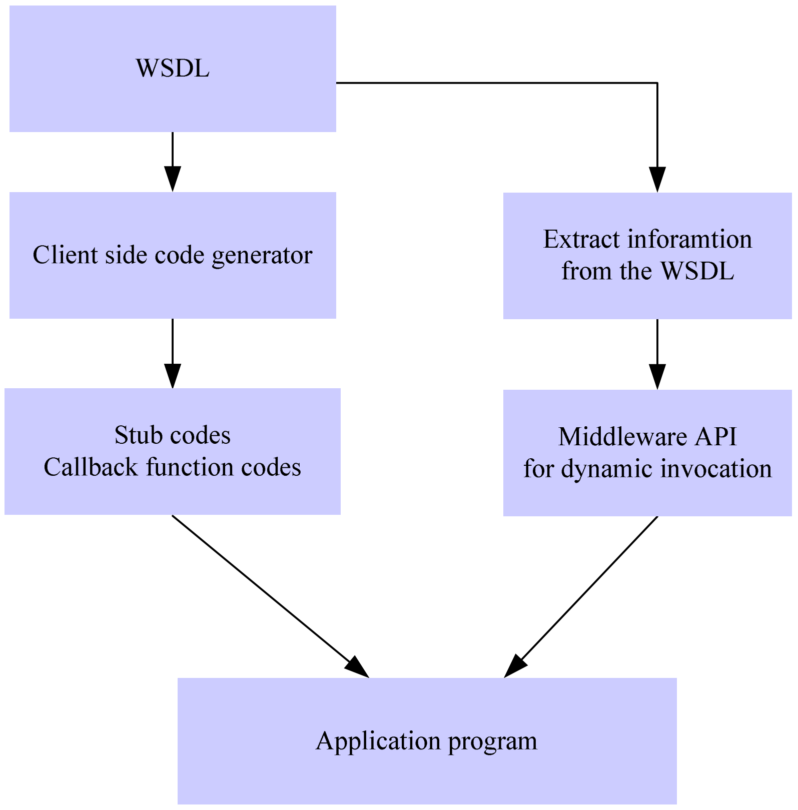

WSDL
Client side code generator
Extract inforamtion from the WSDL
Stub codes Callback function codes
Middleware API for dynamic invocation
Application program

Figure 3-11 The procedure of consuming a web service

Without using a client stub, we can invoke a web service with a dynamic method. The dynamic invocation of a web service needs some information from the WSDL. WSDL describes a *types* tag that are sent and received during the invocation, a *message* tag that includes one type, a *portType* tag that describe the forms of operations, a *binding* tag that indicates a communication method, and a *service* tag that displays the location of the service. Figure 3-12 describes which information is needed when a web service is invoked dynamically. WSDL-based dynamic invocation function requires five arguments: a name of WSDL document, a name of an operation, a service location, target name space, and an argument of an operation which is a request message. As seen in figure 3-12, target name space, the names of operations, and the service location can be found in the WSDL. To make a request SOAP message, information of types and message on WSDL is used. The dynamic invocation of a web service is implemented with AXIS2.

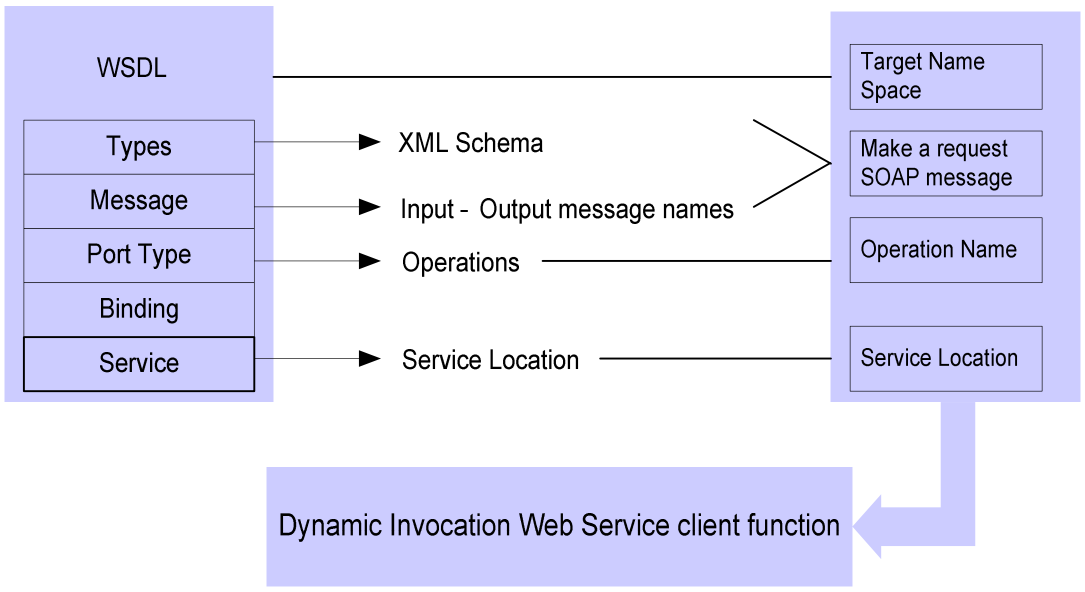


Figure 3-12 WSDL-based dynamic invocation of a web service with AXIS2

The dynamic invocation web service client function returns a response of the web service. The client function consists of an operation client, a request message, and execution of the operation client. Initially a SOAP message is returned into the client function, but it filters the SOAP message to get the body context which is a response message.

3.6.2. Integration of DEVS simulator services

As seen in the figure 3-13, DEVS Simulator Services Integrator is graphic user interface (GUI) which consists of five functionalities: a WSDL handler, a title of integration, a DEVS service handler, a coupling handler, and writing a XML document. The WSDL handler saves on a specific place a selected WSDL which is used in the integration. If a DEVS simulator service is known, we can get WSDL of the service with the WSDL handler. In the GUI, there are three components to enter an address of WSDL and save WSDL. A *textfield* component is where an address of WSDL is written. After the WSDL name of DEVS service is written in the *textfield* by the URL label, clicking the *save WSDL* button writes the WSDL from the URL address on a file. The title of integration provides a file name of a XML document.

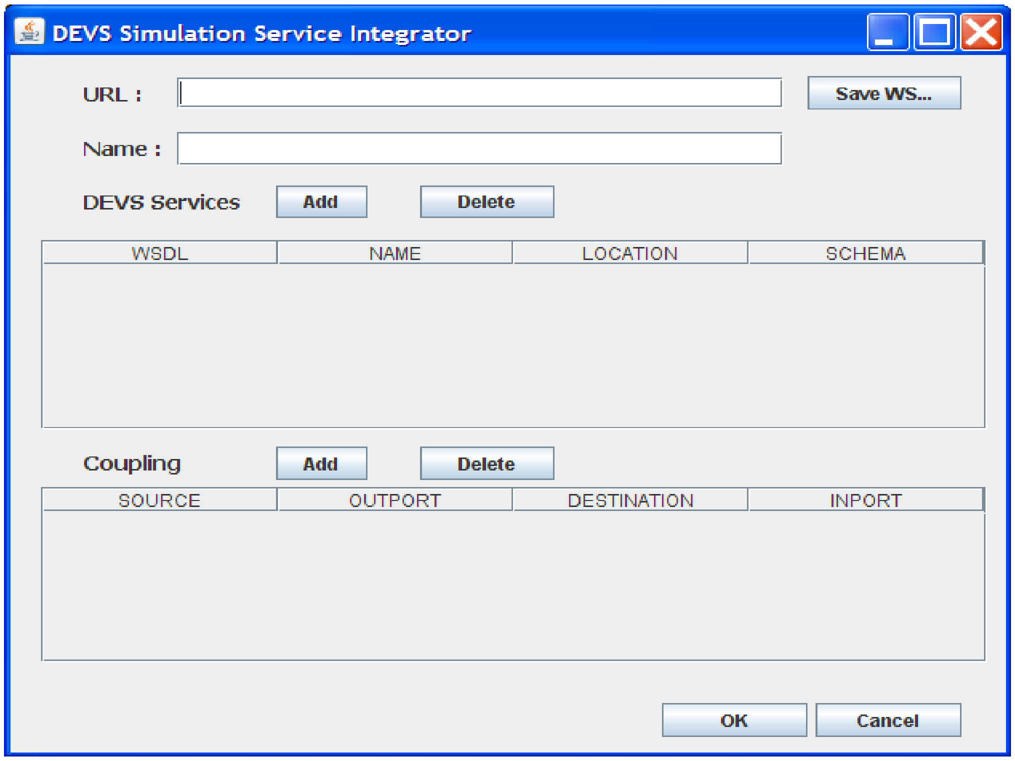


Figure 3-13 DEVS Simulator Services Integrator

The DEVS service handler begins with clicking the “ADD” button by a DEVS services label. The information of service GUI shows up immediately as seen in the figure 3-14. The GUI displays a repository where WSDLs are saved if “Show” button is clicked. The user selects a WSDL file which is used in the integration. The selected WSDL from the repository provides a WSDL file name, a model name, WSDL location, and schema location obtained by invocation of an operation called *getSchemaInfo* in the figure 3-7. That information is displayed on the table below the “DEVS services“ label.

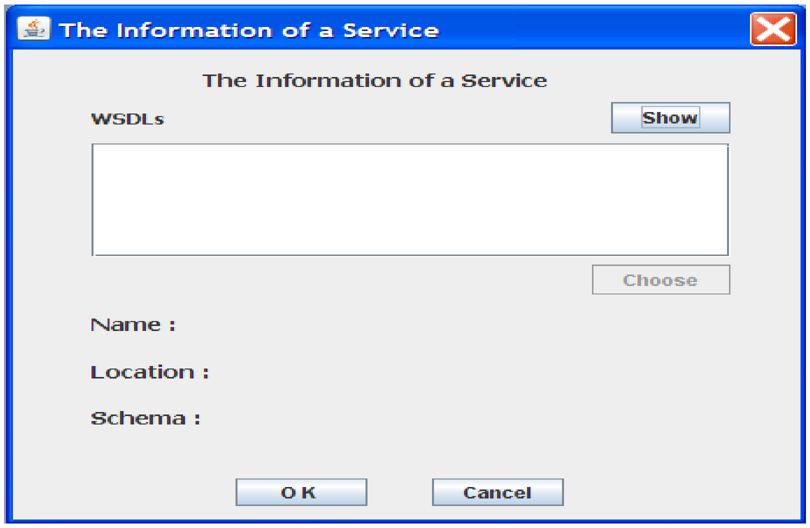


Figure 3-14 The Information of a service

After selecting some DEVS simulator services in the DEVS service handler, the coupling handler is carried out. Pushing the “ADD” button by the “Coupling” label shows a GUI for helping make coupling between DEVS simulator services as seen in the figure 3-15. The coupling GUI displays a source, an output message, a destination, and an input message if the output message is matched to the input message. When displaying an output message and an input message, the invocations of three operations are performed to get the name of the output ports and input ports and the message type of each port. The operations are *getInports*, *getOutports*, and *getType* in the figure 3-7. The coupling information is shown in the table below the “Coupling” label.

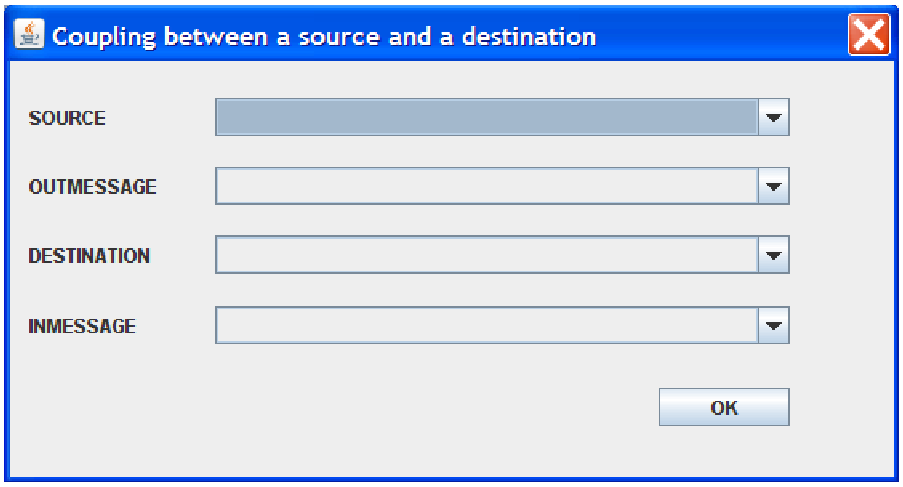


Figure 3-15 Coupling GUI

After finishing the integration of DEVS simulator services, clicking the “OK” button creates a XML document structured to contain the information from the integrator. The schema for the XML document is defined to validate an instance of the schema. The XML document begins with a devswsintegrator tag, and has five tags, that is, *title*, *services*, *couplinginfo*, *inports*, and *outports*. The *services* tag can have many *model* tags which have *wsdl*, *name*, *location*, and *schema* tags. Similarly, *couplinginfo* tag can have many *coupling* tags which have *source*, *outport*, *destination*, and *inport* tags. The *devswsintegrator.xsd* is the following.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" >
	<xs:complexType name="inport">
		<xs:sequence>
			<xs:element name="inport" type="xs:string"
maxOccurs="unbounded"/>
		</xs:sequence>
	</xs:complexType>
	<xs:complexType name="outport">
		<xs:sequence>
```

```
                    <xs:element name="outport" type="xs:string"
maxOccurs="unbounded"/>
                </xs:sequence>
        </xs:complexType>
        <xs:complexType name="modelinfo">
                <xs:sequence>
                    <xs:element name="wsdl" type="xs:string" />
                    <xs:element name="name" type="xs:string" />
                    <xs:element name="location" type="xs:string" />
                    <xs:element name="schema" type="xs:string" />
                </xs:sequence>
        </xs:complexType>
        <xs:complexType name="couplings">
                <xs:sequence>
                    <xs:element name="source" type="xs:string" />
                    <xs:element name="outport" type="xs:string" />
                    <xs:element name="destination" type="xs:string" />
                    <xs:element name="inport" type="xs:string" />
                </xs:sequence>
        </xs:complexType>
        <xs:complexType name="coupling">
                <xs:sequence>
                    <xs:element name="coupling" type="couplings"
maxOccurs="unbounded"/>
                </xs:sequence>
        </xs:complexType>
        <xs:complexType name="model">
                <xs:sequence>
                    <xs:element name="model" type="modelinfo"
maxOccurs="unbounded"/>
                </xs:sequence>
        </xs:complexType>
        <xs:element name="devswsintegrator">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="title" type="xs:string" />
                        <xs:element name="services" type="model" />
                        <xs:element name="couplinginfo"
        type="coupling" />
                        <xs:element name="inports" type="inport" />
                        <xs:element name="outports" type="outport" />
                    </xs:sequence>
                </xs:complexType>
        </xs:element>
</xs:schema>
```

3.6.3. Execution of integrated DEVS simulator services

Execution of integrated DEVS simulator services consists of two parts. One is to prepare the simulation, and the other is simulation. The preparation of the simulation includes making instances of client proxies for DEVS simulator services and structuring coupling information with a XML document. To easily handle the XML document, JAXB is used as seen in the figure 3-16. JAXB generates Java classes containing schema tags through compilation of the schema and converts the XML document to Java instances. From the Java instances, we can extract the information of all tags that is used to generate client proxies and to make a data structure for coupling information.

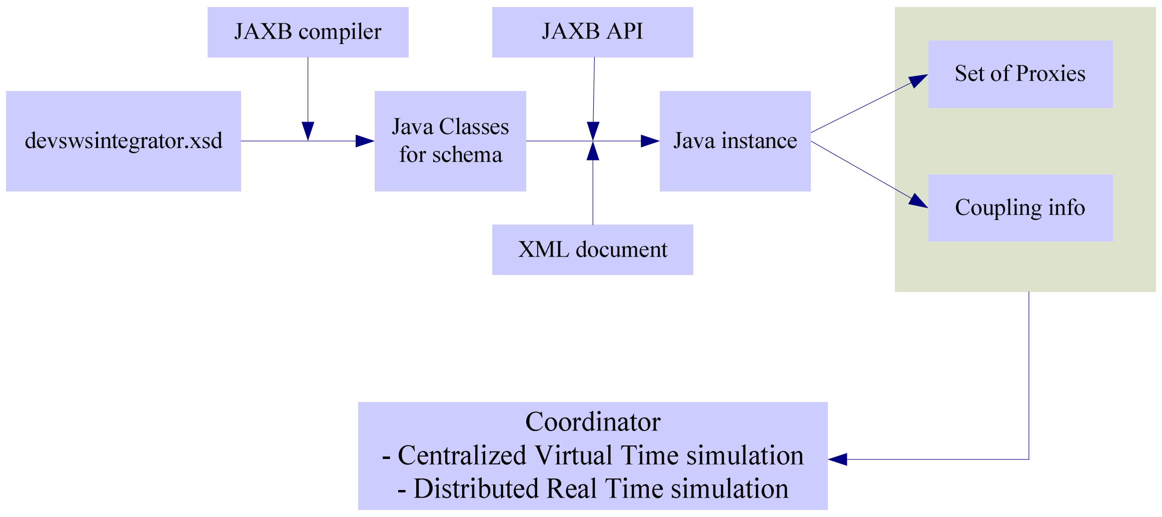


Figure 3-16 The procedure of preparing simulation

The execution of the simulation adopts a centralized virtual time simulation method which controls simulation protocols in the coordinator and distributed real time simulation. Figure 3-17 represents the centralized virtual time simulation protocol that displays calling operations in the coordinator to the DEVS simulator services. The coordinator has the instances of client proxies for the DEVS simulator services. The simulation begins with calling an *initialize* operation to all simulator services. After that, the coordinator requests a *getTN* operation to get next time for an internal event of a

DEVS model. Calculating a minimum time of next times, a *lambda* operation is called with an argument which is the minimum time. After the *lambda* operation, the simulator service with minimum next time produces an output message. The response of the *getOutput* is output messages of the simulator services. The coordinator looks up the coupling information which displays the flow of messages to inject the output message to a corresponding simulator service. Through a *receiveInput* operation, the output message is sent to the simulator service which has the corresponding DEVS model. After routing the output message, the coordinator requests a *deltFunc* operation to all simulator services to execute an internal or external event function in the simulator service. The coordinator repeats this procedure except the initialization of the simulator service until meeting a certain condition to terminate the simulation.

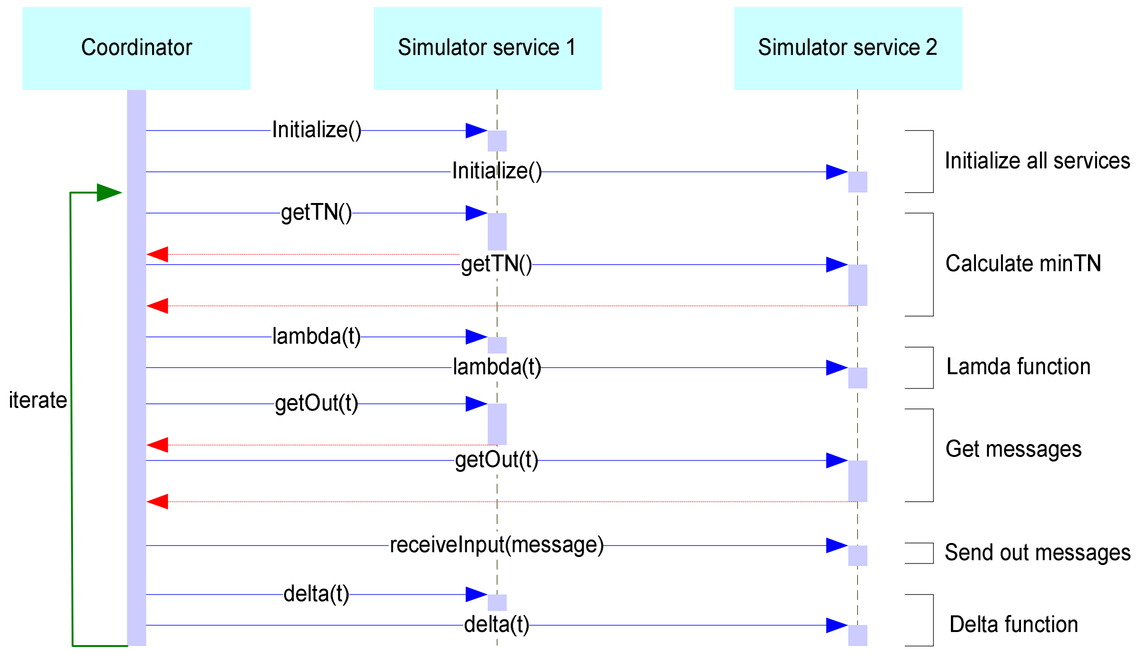


Figure 3-17. The centralized simulation protocol

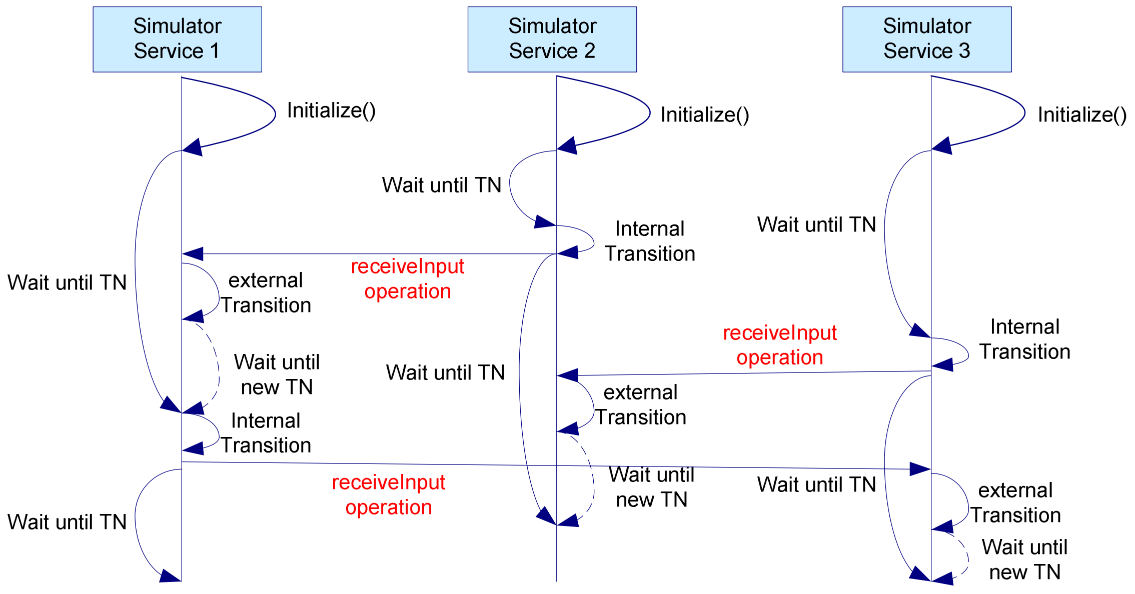


Figure 3-18 The decentralized real time simulation protocol

Figure 3-18 depicts decentralized real time DEVS simulation protocol which starts with the initialization of the DEVS models in the *RTSimulators* that the DEVS simulator service provides for the decentralized real time simulation. Each *RTSimulator* waits for the internal event time (*TN*) after which internal transition occurs. If one of the *RTSimulators* has wall-clock time equal to *TN*, the *RTSimulator* produces an output message, sends the output message to the *RTSimulator* with the corresponding DEVS model according to the coupling information, and executes a delta function which rearranges the state and the internal event time of the DEVS model. Server2, figure 3-18, shows "*send out message*" after internal transition and wait again with *TN* regenerated by the delta function. Meanwhile, Server1 receiving a message from the Server2 executes an external transition function included by the delta function and recalculates *TN* to wait. The interaction between the server2 and the server1 does not affect the server3 which waits for its internal event time. The decentralized real time simulation is terminated when the internal event time of each DEVS model goes to infinity.

3.7. DEVS message to XML message

DEVS messages are defined as pairs consisting of a port and a value in the DEVS modeling and simulation. The DEVS implementations of the DEVS theory use the pairs to express DEVS messages. That means that the DEVS messages can be converted to a common expression in the XML. We design a common XML message to cover generic DEVS messages.

```
<Message>
  <content>
    <port> port name</port>
    <entity>
       <class> class name </class>
           < variable name type = variable type> value </variable name>
           .
           .
     </entity>
  </content>
  <content>
.
.
</Message>
```

Figure 3-19 The structure of the XML message

Figure 3-19 represents the structure of the XML message starting with a *Message* tag. The *Message* tag consists of *content* tags whose elements are a *port* and an *entity* tag. The *port* tag contains the name of the port through which messages are sent. The *entity* tag expresses any object as a message used in the DEVS model. It has a *class* tag containing the name of the object. Tags under the *class* tag are created according to the number of variables of the object. The tags have an attribute called *type* describing the type of the variable.

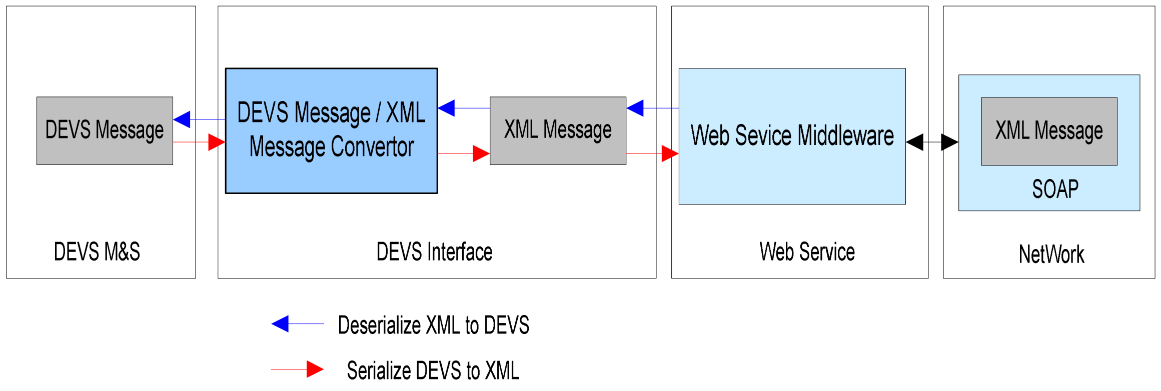


Figure 3-20 The DEVS message and XML message in the web service.

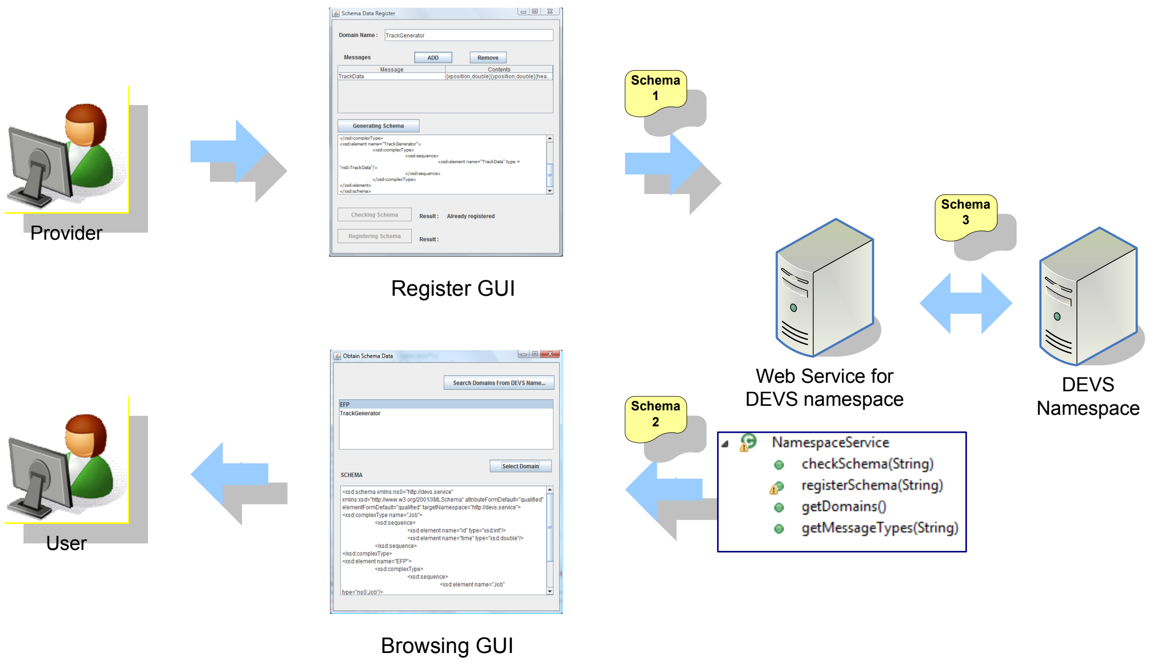
Figure 3-20 represents conversion of DEVS messages to XML messages and vice versa. A DEVS simulator service consists of DEVS modeling and simulation (DEVS M&S), DEVS interface, and web service. The DEVS M&S handle the DEVS messages, and the DEVS interface converts DEVS messages to XML messages, and the web service generates an SOAP message including the XML messages. This procedure is called serialization. The opposite procedure converts XML messages to DEVS messages. It is called deserialization.

CHAPTER 4. IMPLEMENTATION OF THE DEVS NAMESPACE AND DEVS SIMULATOR SERVICES

DEVS modeling and simulation is implemented with Java, C++, or C# languages according to intention of the designers. To demonstrate the concept of interoperability using DEVS Simulators, two DEVS M&S instances implemented with different computer languages and system environments are used. One is DEVSJAVA [10] developed with Java language by ACIMS lab, and the other is ADEVS [11] embodied with C++. In this chapter, we describe the implementation of DEVS namespace, how to create a DEVS simulator service with two different DEVS implementations based on the previous chapter, and XML to the DEVS message conversion method.

4.1. Implementation of the DEVS Namespace

We created a web service called *NamespaceService* through which Schema of a DEVS simulator service is registered and browsed. Figure 4-1 illustrates a procedure of registering and browsing a schema used in a DEVS simulator service. A service provider has responsibility of registration of a schema. When the provider registers the schema, the provider uses a GUI called schema data register. The GUI has client codes for *NamespaceService* web service, which can help easily invoke operations. It displays the response of the operations. Any web service provider who uses a Java based environment or .Net based environment can use the GUI to register a schema. If a user wants to browse the DEVS namespace, the user can use a browsing GUI consisting of two parts. One part is to display all data types in the DEVS namespace and the other part is to show the schema document corresponding to the data type chosen by the user.



Domain Name : TrackGenerator

Message	Contents
TrackData	{xposition,double}{yposition,double}{hea...

```
</xsd:complexType>
<xsd:element name="TrackGenerator">
            <xsd:complexType>
                        <xsd:sequence>
                                    <xsd:element name="TrackData" type =
"ns0:TrackData"/>
                        </xsd:sequence>
            </xsd:complexType>
</xsd:element>
</xsd:schema>
```

Result : Already registered

Result :

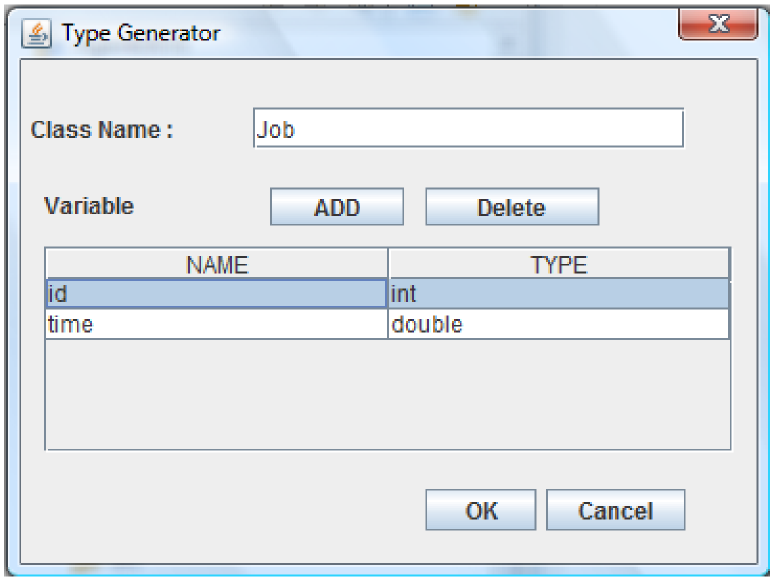
EFP
TrackGenerator

```
<xsd:schema xmlns:ns0="http://devs.service"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" attributeFormDefault="qualified"
elementFormDefault="qualified" targetNamespace="http://devs.service">
<xsd:complexType name="Job">
            <xsd:sequence>
                        <xsd:element name="id" type="xsd:int"/>
                        <xsd:element name="time" type="xsd:double"/>
            </xsd:sequence>
</xsd:complexType>
<xsd:element name="EFP">
            <xsd:complexType>
                        <xsd:sequence>
                                    <xsd:element name="Job"
type="ns0:Job"/>
```

Figure 4-1 Overview of registering and browsing schema

4.1.1. The GUI for schema data registration

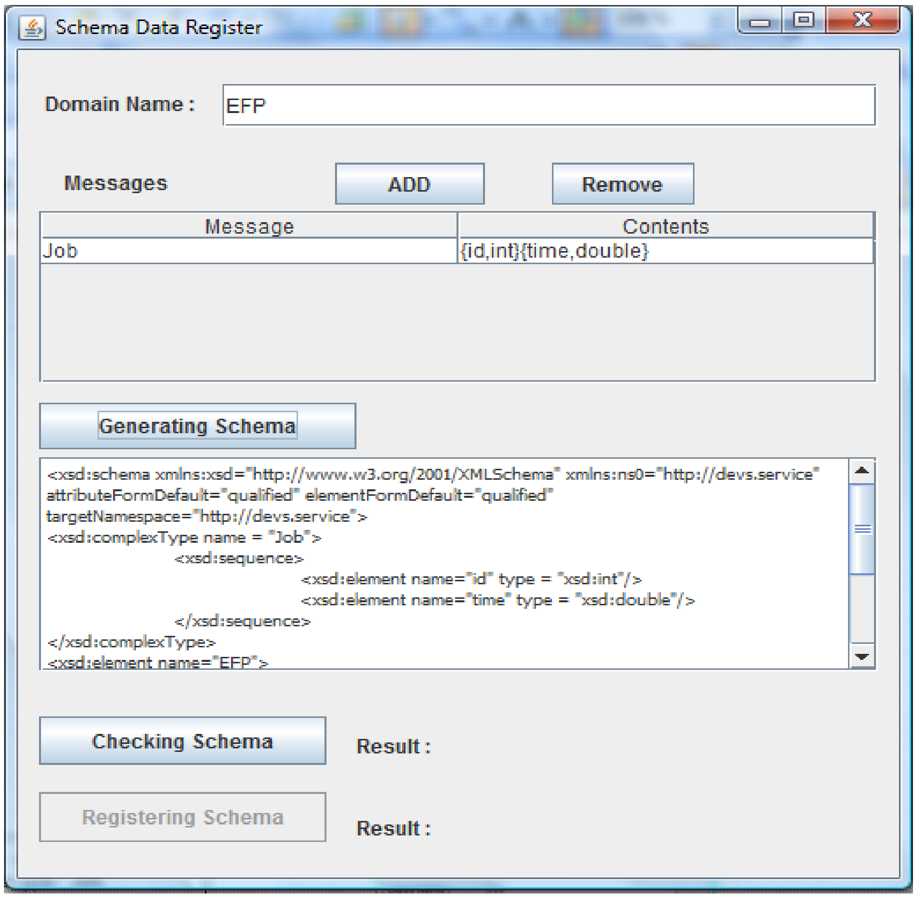
The GUI has three functions: to enter message information such as class name, method name and type, to compose a schema document, and to check and register the schema to DEVS namespace. A service provider can use this GUI to make sure that the schema of DEVS message is registered or to register the schema into the DEVS namespace. If a name of DEVS message is “Job” and the “Job” message has two variables called “id” and “time” whose types are *int* and *double*, respectively, the provider provides information of DEVS messages including a domain name which represents a name of a DEVS model. Figure 4-3 represents the result of conversion “Job” message to a schema. The table on figure 4-3 has two columns, *Message* and *Contents*, that display the messages using the domain. There are two buttons called “ADD” and “Remove” to add or remove a row in the table. A provider adds DEVS messages through an “ADD” button which makes a type generator GUI, as shown. As seen in the figure 4-2, the type generator represents a DEVS message with information of a class name, variable names and types. When the provider finishes entering the information of the DEVS message, a schema document is created and displayed by clicking the button called “Generating Schema”. In figure 4-3, we can see a schema document containing “EFP” domain name, “Job” class name and all names and types of variables in the “Job” class.



Class Name : Job

NAME	TYPE
id	int
time	double

Figure 4-2 The GUI for type generator



Domain Name : EFP

Message	Contents
Job	{id,int}{time,double}

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:ns0="http://devs.service"
attributeFormDefault="qualified" elementFormDefault="qualified"
targetNamespace="http://devs.service">
<xsd:complexType name = "Job">
                <xsd:sequence>
                                <xsd:element name="id" type = "xsd:int"/>
                                <xsd:element name="time" type = "xsd:double"/>
                </xsd:sequence>
</xsd:complexType>
<xsd:element name="EFP">
```

Result :

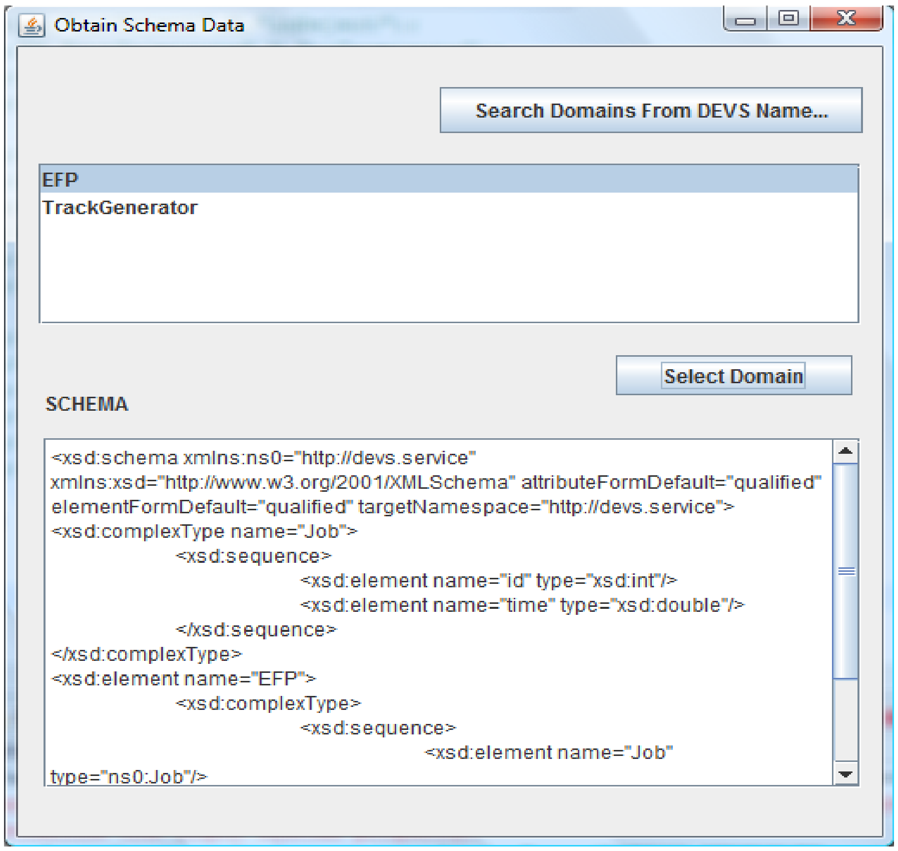
Result :

Figure 4-3 The Example of the GUI for schema register

The “Checking Schema” button makes a *checkSchema* operation in *Namespace-Service* web service invoked with a schema document, and gets the return value which is a Boolean type. If the return value is *true*, the schema is already registered. If *false*, the schema needs to be registered in the DEVS namespace. In case the return value of the *checkSchema* operation is false, the “Registering Schema” button gets enabled and the schema is registered by clicking the button. In this case, a *registerSchema* operation is invoked.

4.1.2. Browsing GUI

Figure 4-4 represents the browsing GUI which has two buttons called “Search Domains from DEVS Namespace” and “Select Domain”. If a user clicks the “Search Domains from DEVS Namespace” button, the GUI shows all domains in the list under that button. When the user selects one of the domains and clicks the “Select Domain” button, then the GUI shows the schema document for the selected domain.



```
<xsd:schema xmlns:ns0="http://devs.service"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" attributeFormDefault="qualified"
elementFormDefault="qualified" targetNamespace="http://devs.service">
<xsd:complexType name="Job">
		<xsd:sequence>
			<xsd:element name="id" type="xsd:int"/>
			<xsd:element name="time" type="xsd:double"/>
		</xsd:sequence>
</xsd:complexType>
<xsd:element name="EFP">
		<xsd:complexType>
			<xsd:sequence>
				<xsd:element name="Job"
type="ns0:Job"/>
```

Figure 4-4 The Example of the GUI for schema browser

4.1.3. *NamespaceService* web service

NamespaceService web service is designed to check, register, browse, and get a schema into DEVS namespace. There are four operations in the service. They are called "checkSchema", "registerSchema", "getDomains", and "getMessageTypes". The "checkSchema" and "registerSchema" are used to check and register a schema document. Both operations have one argument and one return value, which are a string type and Boolean type, respectively. The "checkSchema" operation extracts the first element of a schema, called a domain name, and checks if the domain name is on the DEVS namespace document. If the name is on the DEVS namespace, the "checkSchema" returns true. If not, the "checkSchema" returns false. The "registerSchema" operation adds the schema document to the DEVS namespace. If there is no error, then the operation returns true. If there is an error during addition of the schema, the operation returns false.

The "getDomains" and "getMessageTypes" are used to browse and get a schema document. The "getDomains" operation has no argument and a string array for a return type. The string array contains all domain names in the DEVS namespace. The "getMessageTypes" has a string as an argument and a string as a return value. The return value contains a schema document for the argument.

4.2. Simulator Services encapsulating DEVSJAVA

4.2.1. DEVSJAVA

DEVSJAVA consists of three libraries, that is, DEVS M&S supporting data structures, modeling, and simulation. All libraries follow an object oriented design concept which presents inheritance, polymorphism, and information hiding. The modeling library is used to create two kinds of DEVS models which are atomic and coupled models. The entity class is a base class for all modeling and DEVS M&S supporting data structure classes as seen in the figure 4-5 [25]. The *devs* class has basic methods constructing DEVS models which are divided to atomic class and coupled class. The digraph class inherits from coupled class and has a container class which stores all *devs* class components in the coupled model.

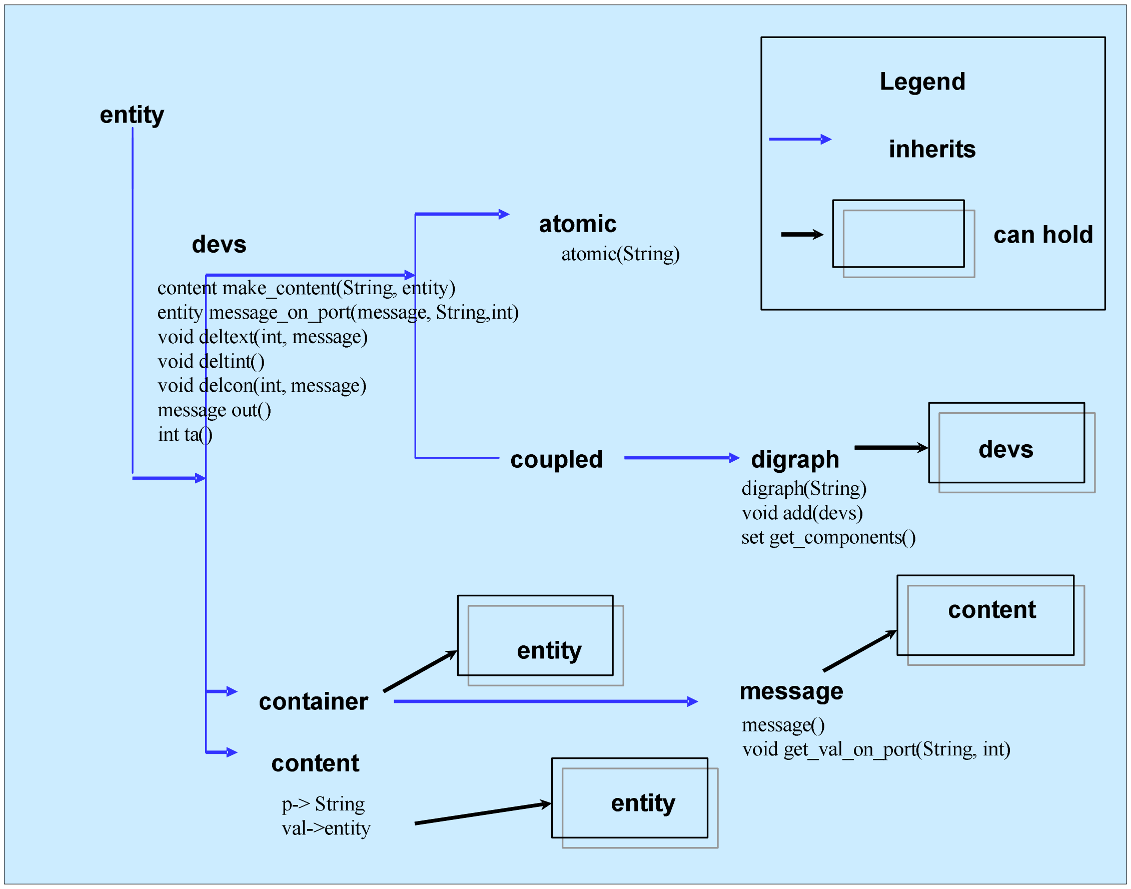


Figure 4-5 DEVSJAVA class hierarchy

The container class is used to handle multiple messages in and out to or from the DEVS model. The message class inherited from the container class contains the set of content classes which consist of a port name and an entity class.

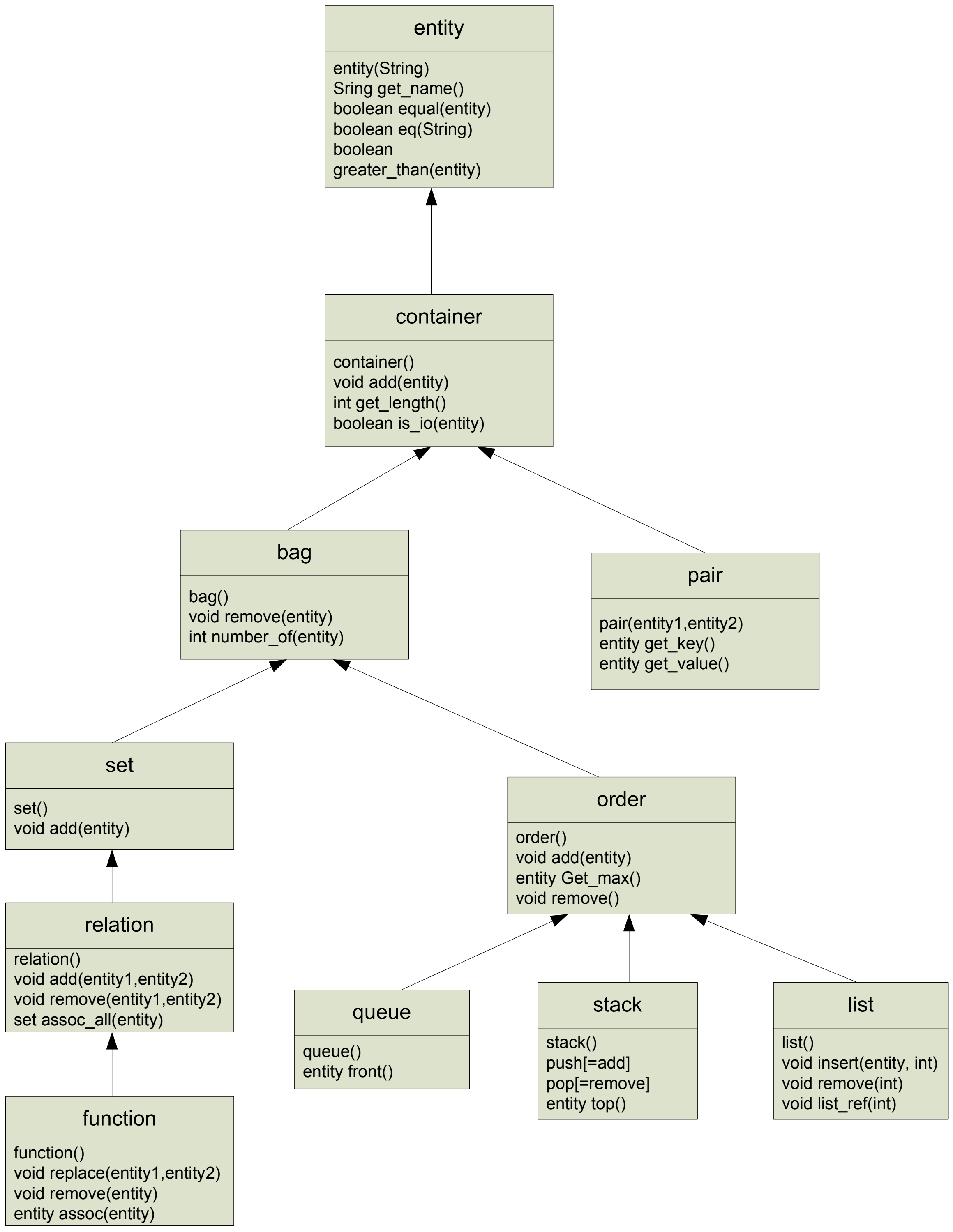


Figure 4-6 Class hierarchy of container class

The data structures support modeling and simulation with holding necessary objects such as models, simulators, and messages. Figure 4-6 shows the container class hierarchy and their main functions. The class is roughly characterized as follows.

- *entity* - the base class for all classes of objects to be put into containers
- *pair* - holds a pair of entities called key and value
- *container* - the base class for *container* classes, provides basic services for the derived classes
- *bag* - counts numbers of object occurrences
- *set* - only one occurrence of any object is allowed in.
- *relation* - is a set of key-value pairs, used in dictionary fashion
- *function* - is a relation in which only one occurrence of any key allowed
- *order* - maintains items in given order
- *queue* - maintains items in first-in/first-out (FIFO) order
- *stack* - maintains items in last-in/first-out (LIFO) order
- *list* - maintains items in order determined by an insertion index

The simulation library helps execute DEVS models. There are two main classes called a coordinator and a simulator in the simulation library. The simulator controls an atomic model, and the coordinator manages the simulators through the message passing such as simulation time and DEVS messages. In case of a DEVS coupled model, it has a hierarchical structure of DEVS components and each component is encapsulated by a coordinator or a simulator according to the type of components. A coupled model is assigned to a coordinator and an atomic model is assigned to a simulator. A top level coupled model is assigned to a top level coordinator which decides minimum time advance from all coordinators and simulators which send time advance and controls simulation protocols. Figure 4-7 represents assignment of a simulator or a coordinator to each model in a top level coupled model [13].

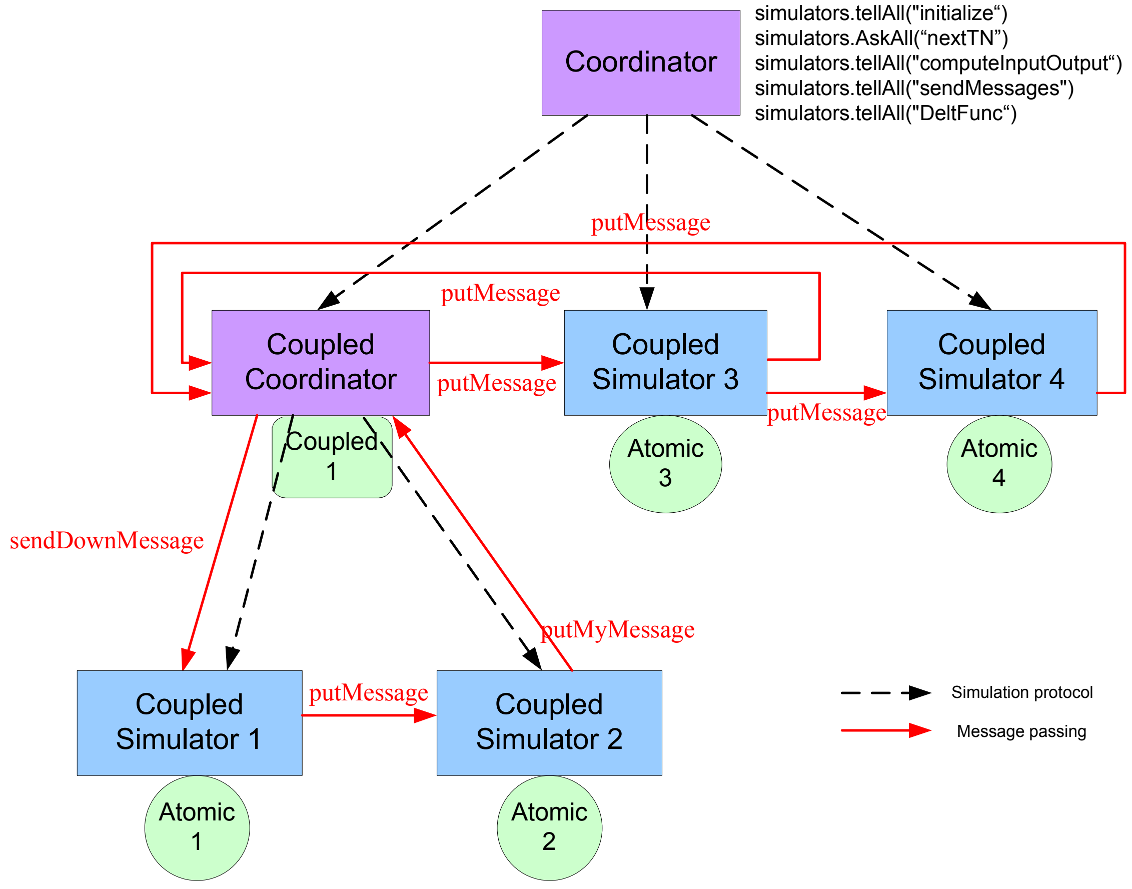


Figure 4-7. The view of relationship between a model and a simulator or a coordinator

The simulation of DEVS model starts with calling a *simulators.tellAll(“initialize”)* function at the top level coordinator as seen in figure 4-5. If a coupled coordinator receives a “*initialize*” message, it calls the same function as that called in the coordinator to its coupled simulators. Dotted lines display propagation of the simulation protocol to all coordinators and simulators. After finishing the *simulators.tellAll(“initialize”)*, the coordinator invokes a *simulators.AskAll("nextTN")*. All coupled simulators get the *nextTN* from an atomic model embedded in them. In case of a coupled coordinator, it propagates a simulation protocol to its coupled simulators and gets the *nextTN*s from the coupled simulators. It sends minimum *nextTN* to the coordinator. The coordinator selects minimum *nextTN* from all *nextTN*s, and calls a *simulators.TellAll("computeInput-Output")* to low level coupled coordinators and simulators with the *minTN*. Each coupled simulator generates and stores an output message. When it receives "*sendMessage*" from the coordinator or the coupled coordinator, it sends the output message to other coupled coordinators or simulators using a "*putMessage*" method. In case of different level message passing, there are two functions to send messages to a lower level or an upper level, they are, "*sendDownMessage*" and "*putMy-Message*". After all output messages are placed according to the coupling information, the coordinator invokes a *simulators.Tell-All("deltfunc")* to make all simulators process internal or external events. This is one cycle of DEVSJAVA simulation which is terminated in meeting an ending condition.

DEVS message can be any object inherited by an entity class. A container for DEVS message is a content class which has two variables representing a port of a model and a DEVS message. To cover multiple content classes, there is a container for contents which is called a message class. More information about DEVSJAVA is in the [27].

4.2.2. DEVSJAVA interface

DEVSJAVA is specific implementation of DEVS theory with the Java language. To adept DEVSJAVA models to DEVS simulator service, interfacing classes called

DEVSJAVA interface are required. DEVSJAVA interface includes a *Simulator* class, an *Atomic* class, a *Digraph2Atomic* class, and a Message class which has three functions for DEVS message converting. The DEVSJAVA interface helps DEVS simulator service extract information from the DEVSJAVA model and convert DEVSJAVA messages to messages compatible to the web service.

Modeling interface is an Atomic and a *Digraph2Atomic* class. The *Atomic* class has a DEVSJAVA atomic class and basic atomic DEVS functions to provide information of the DEVSJAVA atomic model. The *Digraph2Atomic* class is designed to represent a coupled model as an atomic model because the DEVS simulator service provides atomic model functions. DEVS modeling is *closure* under coupling which means behaviors of a coupled model are expressed to behaviors of one atomic model.

```
coordinator coord
public void initialize() {
      coord.initialize();
}
public void deltext(double e, message x) {
      coord.simInject(e, x);
}
public void deltint() {
      coord.wrapDeltfunc(coord.getTN());
}
public message out() {
      coord.computeInputOutput(coord.getTN());
      return (message)coord.getOutput();
}
public double ta() {
      return coord.getTN() - coord.getTL();
}
```

Figure 4-8 The atomic model functions with a coordinator embedding a coupled model

Figure 4-8 shows a part of codes of the *Digraph2Atomic* class which represent an atomic model with a coordinator class coming from the DEVSJAVA simulation package. When converting a coupled model to an atomic model, the *Digraph2Atomic* extracts input ports and output ports from the coupling information of the coupled model to provide ports to the atomic model. Each function representing an atomic model is implemented by functions provided by a coordinator. For example, *initialize, deltext,* and *deltint* functions contain *initialize, simInject,* and *wrapDeltafunc* functions from a coordinator class, respectively. A *ta* function contains *getTN* and *getTL* functions from the coordinator class. An *out* function is expressed with *computeInputOutput* and *getOutput* functions

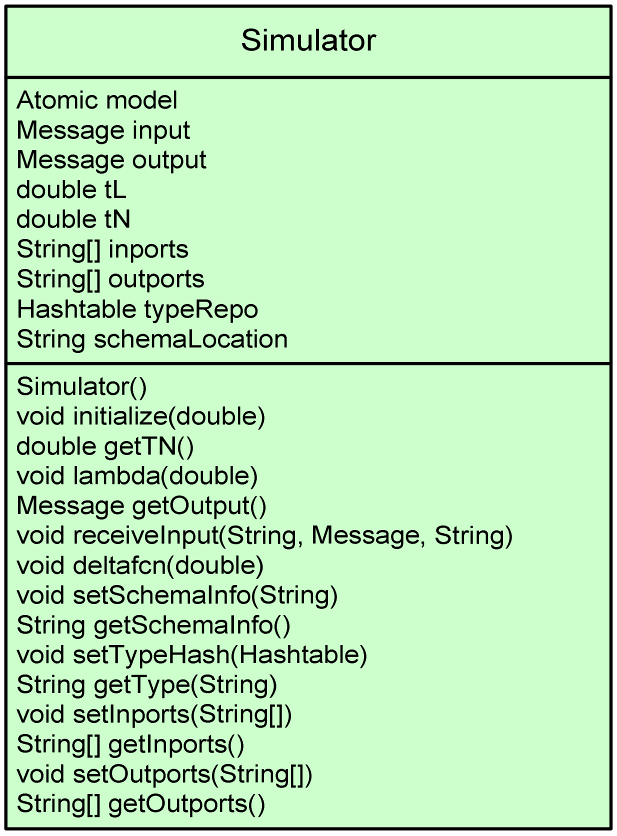
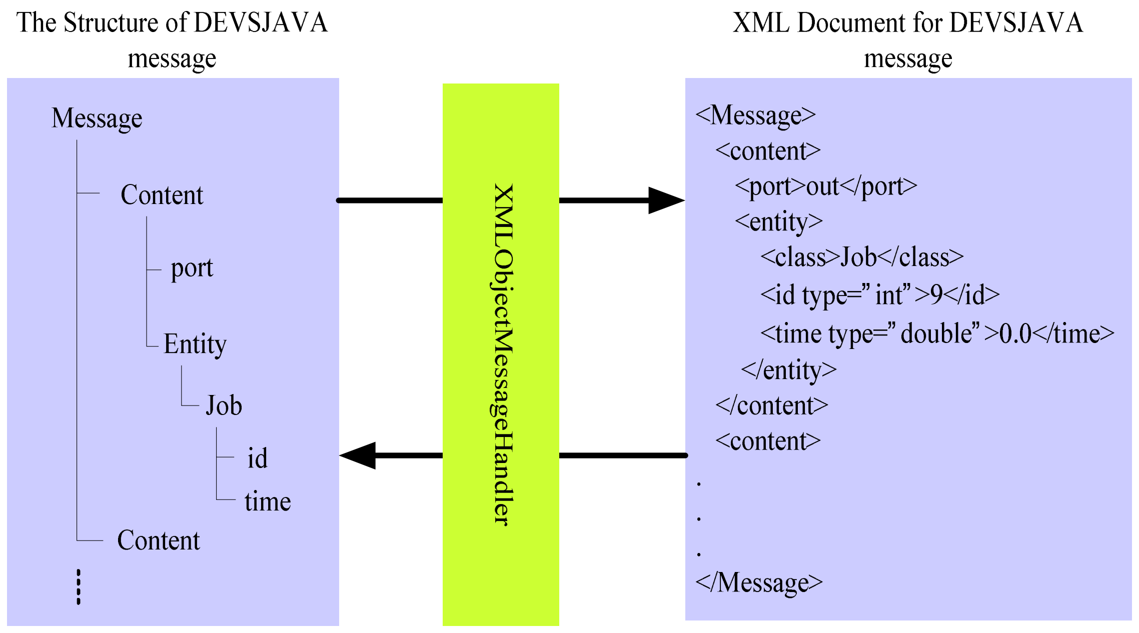


Figure 4-9 Simulator class view

The *Simulator* class consists of basic atomic model functions, schema and ports information function, and port type information functions. Figure 4-9 represents a

Simulator class which has an *Atomic* class, two *Message* classes, and two arrays of *String*. The *Atomic* class can represent an atomic DEVS model or a coupled DEVS model using a *Digraph2Atomic* class. The *Message* classes are used to handle input messages and output messages. The arrays of *String* are for the names of input and output ports. Also, the *Simulator* class has a *Hashtable* and a string variable for location of schema. The *Hashtable* contains ports and type information as keys and values. With a *getType* method, the type information of the port is obtained from the *Hashtable*. Therefore, the *Simulator* class has methods to handle DEVSJAVA model and schema information of ports, and connects the information of DEVSJAVA models to the DEVS simulator service.


The Structure of DEVSJAVA message
Message
Content
port
Entity
Job
id
time
Content
XMLObjectMessageHandler
XML Document for DEVSJAVA message

```
<Message>
  <content>
    <port>out</port>
    <entity>
      <class>Job</class>
      <id type="int">9</id>
      <time type="double">0.0</time>
    </entity>
  </content>
  <content>
.
.
.
</Message>
```

Figure 4-10 Example of XML Object Message Handler

The *Message* class converts a DEVSJAVA message class to XML message and vice versa using an *XMLObjectMessageHandler* class. Figure 4-10 depicts the example of conversion XML message to DEVS message and vice versa using *XMLObjectMessage-Handler*. On the left side, the structure of DEVSJAVA message consists of multiple Content classes which have a name of port and an entity class. In this example, a *Job* class inheriting an entity class has two variables called id and time. The *XMLObjectMessageHandler* takes a DEVS message as an input to generate a XML message. The right side on the figure 4-8 displays the XML message for the DEVS message. The tags follow the structure of the DEVS message. In the entity tag, there are three tags called class, id, and time for a specific Job class example. The class tag indicates a class name and the id and time tags have an attribute called type representing a data type of the variable. The message tag can include multiple content tags to express that DEVS models can get multiple messages during the simulation.

Figure 4-11 illustrates the algorithm of the conversion of the DEVSJAVA message to the XML message. The algorithm begins with receiving the DEVSJAVA message as an argument. The *msg* in line 1 is a variable for a DEVSJAVA message. The *messageXML* is declared to return the XML message. The document in line 3 represents a XML document from which all XML elements are written. The *messageElement* representing a top tag in the XML message is created from the XML document. From the *msg* in line 1, *contentIterator* can be obtained as seen in line 5. The *iterator* contains a set of content classes. We extract each content class to get the information inside the content in line 7. The *contentElement* is created in line 8 and is appended to *messageElement*. From the content class, a port name is obtained by the *getPortName*() function in the content class. Thereafter, the *portElement* is created, has a text value as the port name, and is appended to the *contentElement*. The *entityElement* is created in line 14 and is appended to the *contentElement*.

```
1.  msg ← DEVS message
2.  messageXML :=””;
3.  document := newDocument();
4.  messageElement :=document.createElement(“Message”);
5.  contentIterator := msg.mIterator();
6.
7.  while(contentIterator != 0)
8.    contentElement := document.createElement(“content”);
9.    messageElement.appendChild(contentElement);
10.   port := content.getPortName();
11.   portElement := document.createElement(“port”);
12.   portElement.setText(port);
13.   contentElement.appendChild(portElement);
14.   entityElement := document.createElement(“entity”);
15.   contentElement.appendChild(entityElement);
16.   classElement := document.createElement(“class”);
17.   object :=content.getValue();
18.   class :=object.getClass();
19.   classElement.setText(class.name);
20.   entityElement.appendChild(classElement);
21.   field[] :=class.getDeclaredFields();
22.
23.   while(field[].size != 0)
24.     variableElement := document.createElement(field.name);
25.     Class t = field.getType();
26.     type :=makeType(t);
27.     variableElement.setAttribute(“type”, type);
28.     Method := getGetMethod(makeGetMethodName(field.name, class, type)
29.     Result := Method.invoke();
30.     variableElement.setText(getStringValue(Result));
31.     entityElement.appendChild(variableElement);
32. messageXML :=document.toString();
```

Figure 4-11 The algorithm of the conversion of the DEVSJAVA message to the XML message

To get the DEVSJAVA model message, the *getValue*() function in the content class is used. The DEVSJAVA model message can be any kind of object. In the Java, we can get the information of variables of any object using the *getDeclaredFields* function in the

Class class in line 21. The field array contains the information of variables, and each element of the field array is recorded to the XML message. Line 24 creates a *variableElement* using the name of the field class. Through lines 25 and 26, we can get the type of the variable. For example, if a variable is *int*, then the type is assigned to “int”. If a variable is *int[]*, then the type is assigned to “intArray”. The type is added to the *variableElement* as an attribute as seen in line 27. We can get the value of the variable using the method provided by the object. We assume that a DEVSJAVA model message class provides get- and set- methods to access the variables. We can invoke a get- method to get the value of the variable through the dynamic method call provided by Java in lines 28 and 29. After getting the value of the variable, the value is added as a text to the *variableElement* which is appended to the *entityElement*. This procedure (lines 23 to 31) is repeated until writing the information of all variables to the XML message. The procedure for the content class (lines 7 to 31) is repeated if the DEVSJAVA message has multiple content classes.

```
<Message>
  <content>
    <port>out</port>
    <entity>
       <class>Job</class>
       <id type=” intArray” >
          <element>1</element>
          <element>2</element>
       </id>
       <time type=” double” >0.0</time>
     </entity>
  </content>
</Message>
```

Figure 4-12 The example of DEVS message with an array

This conversion algorithm for the DEVSJAVA message to the XML message covers the DEVSJAVA model message with primitive types and primitive array types. The DEVSJAVA model message should have get- and set- methods to handle the variables. In case of primitive array types, the XML message needs to have an additional tag called “element”. For example, if *Job* class has one *int* array type variable and one double type variable, the XML message looks like figure 4-12 where a *id* tag includes multiple element tags to contain the elements of the array.

```
cInfo := contentInfo;
contentNode := getContentNode( XML document);
while(contentNode != 0)
  if(portNode)
    cInfo.port := port;
  while(entityNode !=0)
    if(classNode)
      cInfo.classType := class;
    else
      v := vector, variable := NodeName, type := NodeAttribute(“type”);
      value := “”;
      if(type.endswith(“Array”))
        while(ElementNode !=0)
          value := Element value;
          v.add(value);
        cInfo.setArray(type,v);
      else
        value := Node value;
      cInfo.setBag(variable,type,value);
```

Figure 4-13 The algorithm to extract the information of the XML message

To convert the XML message to the DEVSJAVA message, there are two steps which are used to gather the information into the predefined class and a container, and to make an instance of the DEVSJAVA message. Figure 4-13 shows the algorithm to extract the information of the XML message. The *contentInfo* class in line 1 is for gathering the information of the *content* tag. In line 2, we get the nodes for the *content* tags called *contentNodes* from the XML document. The *contentNode* is handled to accumulate the information from the *content* tag. The *contentNode* has a *portNode* and an *entityNode*. The name of a *portNode* is stored in the *contentInfo* class in line 5. To extract the information in the *entityNode*, we search all nodes in the *entityNode*. We define a *class* tag, but others depend on the name of variables. If the *classNode* is encountered, the name of the *classNode* is stored in the *contentInfo* class in line 8. If not, we assume that the tag is one of the variables. There are ramifications to handle primitive variables and primitive array variables between lines 12 and 18. If a type is “array”, values of element tags are extracted, stored in a vector, and a pair consisting of the type and the vector is sent to the *contentInfo* class as seen in lines 15 and 16. If the type is primitive, a value is obtained by the *variableNode*. Finally line 19 shows that the variable, the type, and the value are stored in the *contentInfo* class.

Based on the *contentInfo* class created with the algorithm shown in the figure 4-13, DEVS message is created using the algorithm shown in the figure 4-14. A message and an object class are declared in lines 1 and 2. The message class comes from DEVSJAVA API, and the object class represents DEVSJAVA model message, which is not defined yet. If a *classType* in the *contentInfo* class is “entity”, the object class is replaced to an entity class with a value from the *contentInfo* class as an argument shown in line 4. Line 5 adds an instance of a *content* class to the *message* class. The instance of the content class has a name of a port from the *contentInfo* class and the entity class as the arguments. If the *classType* is not “entity”, the *classType* is reassigned with a class location and the *classType*, and the object class is assigned to a dynamic created instance with the *classType*. To assign variables in the object to values from the *contentInfo* class, a bag for *NTV* classes, which represent names, types, and values of the variables, is extracted in the *contentInfo* class (line 9). The variables of the object class in line 8 are set to the specific values from the *NTV* class using a method class provided by Java (line 14). Line 15 adds a *content* class with a port and the object class to the message class. This algorithm returns an instance of a message class containing the information of XML message.

```
1.  massage msg;
2.  Object o;
3.  If(cInfo.classType = “entity”)
4.      o:=new entity(value);
5.      msg.add(new content(cInfo.port, (entity)o);
6.      continue;
7.  classType := class location + cInfo.classType;
8.  o:= dynamic created instance with classType;
9.  BagNTV := cInfo.bagNTV;
10. While(BagNTV !=0)
11.      variable := NTV.name;
12.      type := NTV.type;
13.      value := NTV.value;
14.     Assign values to variables with a Method class
15. msg.add(new content(cInfo.port, (entity)o));
```

Figure 4-14 The algorithm to make an instance of DEVS message

4.2.3. DEVS simulator service with DEVSJAVA

To create web services for DEVSJAVA, we need to put all things together mentioned in the previous sections such as DEVSJAVA API, DEVS interface, and a class containing operations of the DEVS simulator service.

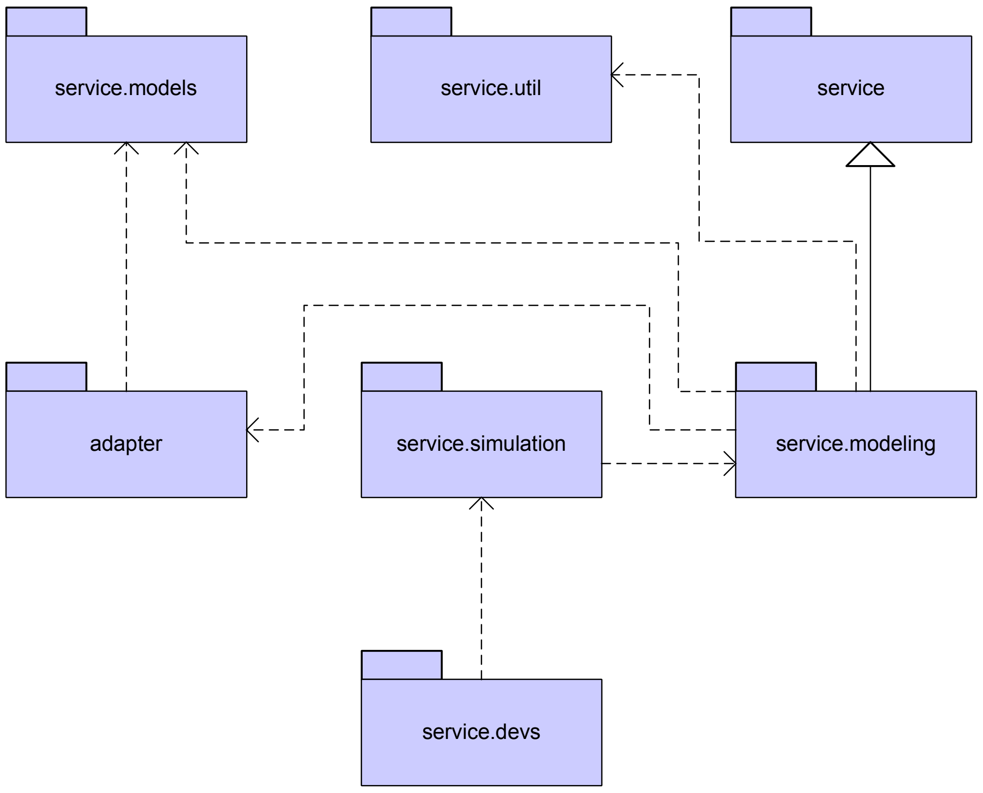


Figure 4-15 The package diagram of the DEVS simulator service with DEVSJAVA

Figure 4-15 depicts the package diagram for the DEVS simulator service with DEVSJAVA. There are seven packages to create the DEVS simulator service. The actual service of DEVSJAVA is in the *service.devs* package where a Java class having all operations of the service is implemented. The DEVSJAVA model is in the *service.models* package. The *adapter* package has a *Digraph2Atomic* class to make a coupled model seen to an atomic model. The *service.modeling* package has classes to connect DEVSJAVA model to DEVS simulator service such as an *Atomic* and a *Message* classes. The *Atomic* class makes an atomic or a coupled DEVS model look like one class type, that is to say, the Atomic class. The *Message* class has an *XMLObjectMessageHandler* class in the *service.util* package and a message class from DEVSJAVA. Message conversion is done in the *Message* class. The *service.simulation* package has a simulator class handling DEVS simulation protocol with the *Atomic* class.

After all classes are implemented, the classes need to be placed in the web server where we use an Apache tomcat6 server and AXIS2 middleware. We can deploy all classes into the specific folder. Another option is to compress all classes as an archive. The archive has a structure to contain all classes and services.xml document which indicates a service class and message exchange patterns for the web service. The message exchange patterns show the shapes of operations. For example, if an operation has an argument and no return type, the message exchange pattern is in-only. If an operation has an argument and return type, the message exchange pattern is in-out.

```xml
<service name="EFModel">
 <description>Please Type your service description here</description>
 <messageReceivers>
   <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-only" class="org.apache.axis2.rpc.receivers.RPCInOnlyMessageReceiver" />
   <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-out" class="org.apache.axis2.rpc.receivers.RPCMessageReceiver" />
 </messageReceivers>
 <parameter name="ServiceClass" locked="false">service.devs.DevsSimulatorTemplet</parameter>
</service>
```

Figure 4-16 An example of a services.xml

Figure 4-16 represents an example of a services.xml. The *services.xml* consists of several tags that implicate their roles. The service tag represents the name of the web service. In figure 4-16, we see that the name of the web service is *EFModel*. The description tag lets users know the information of the web service. The *messageReceivers* tag has multiple *messageReceiver* tags which indicate the message exchange patterns for operations in the web service. The parameter tag shows the location of the service class.

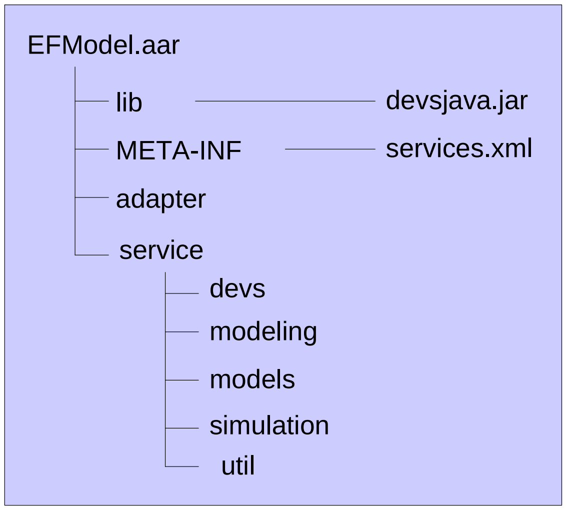


Figure 4-17 The structure of the service archive for an *EFModel* service

Figure 4-17 depicts the structure of the service archive for an example web service called *EFModel* service. The name of this archive is *EFModel.aar* which is the same as .zip or .jar, so we can easily create *.aar* file if using zip or jar programs and changing the file extension to *aar*. The archive should include all server side class files, libraries, and a folder named META-INF with the WSDL and services.xml files. The libraries are used to support the service class.

We need the environment of integrating an Apache web server and AXIS2 to deploy *.aar* file. A web archive (WAR) file is used to connect between the server and AXIS2 and has a specific structure consisting of *axis2-web*, META-INF, and WEB-INF folders. The WAR file contains contents for web services and has a configuration file called a web.xml in the WEB-INF folder. The web.xml contains directions of processing web requests between the web server and AXIS2. The web service compressed to *.aar* is located in the services folder in the WEB-INF folder.

4.3. Web Service encapsulating ADEVS

4.3.1. ADEVS

A discrete event system simulator (ADEVS) is implementation of DEVS based on parallel and dynamic DEVS formalism using C++ [11]. It consists of modeling, simulation, and container libraries as seen in figure 18, which represents the classification of ADEVS API into their usages. The APIs do not have many source codes but they are used in the various domains to solve specific domain problems with a simulation approach.

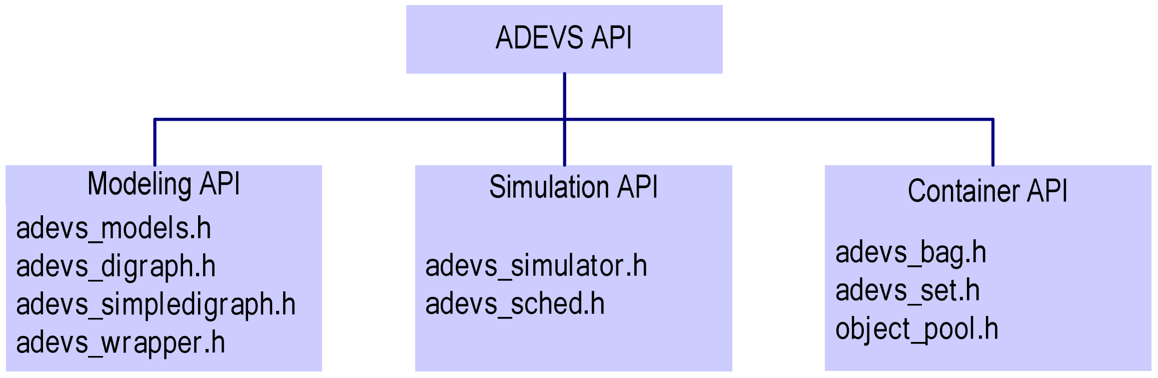


Figure 4-18 The classification of ADEVS header files into their usages

The modeling API is used to create atomic and coupled models containing elements of DEVS formalism such as states, internal/external event handler, output messages, and input/output ports. Figure 4-19 represents hierarchical structures of ADEVS modeling classes. The top level class is the *Devs* class with an X message, which is implemented with templates to accommodate a generic message. The *Devs<X>* class contains basic operations used during simulation to indicate whether the model is an atomic or a couple model and the information of its parent. The *Atomic<X>* class inherits *Devs<X>* class and provides basic functions to implement an atomic DEVS formalism. The *Network<X>* class is a base class for DEVS coupled models and provides *getComponents* and route functions. The *getComponents* function is used to obtain all components in the coupled model, and the route function sends the external message into the component(s) according to the coupling information displaying the flow of the messages in the coupled model.

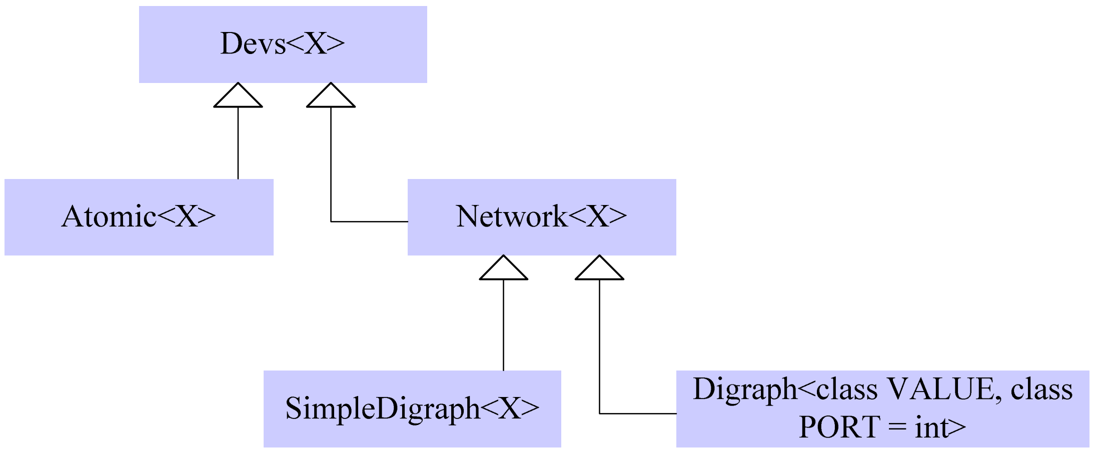


Figure 4-19 Hierarchy structures of ADEVS modeling class

The *SimpleDigraph<X>* and *Digraph<class VALUE, class PORT = int>* have *Network<X>* as parents. The difference between two derived digraphs from Network<X> class is a type of message. The *SimpleDigraph<X>* has a single X message, but *Digraph<class VALUE, class PORT=int>* class has the *PortValue* object to make couplings among the components. The port type in the *PortValue* object is integer as a default. The *Digraph* class has *add* and *couple* functions with the functions from the

Network class. The *add* function is to add a model to the network, and the *couple* function is to connect the source model to the destination model.

The simulation API has a *schedule* class and a *simulator* class which controls simulation protocol. To get the minimum time for the next event, the simulator uses the scheduler with a bag container storing atomic models and gets minimum next event time of atomic models in the scheduler. Therefore, ADEVS does not use a hierarchy structure of model when calculating minimum event time (*TA*). When passing messages from a source model to a destination model, the simulator calls a route function in the simulator class and each coupled model uses its route function in the *Digraph* class. The simulation of an ADEVS model is executed by three functions in the simulator class as seen in figure 4-20.

```
while (nextEventTime() < DBL_MAX)
{
        computeNextOutput();
        computeNextState(bogus_input,sched.minPriority());
}
```

Figure 4-20 Simulation of ADEVS model

The functions are the following.

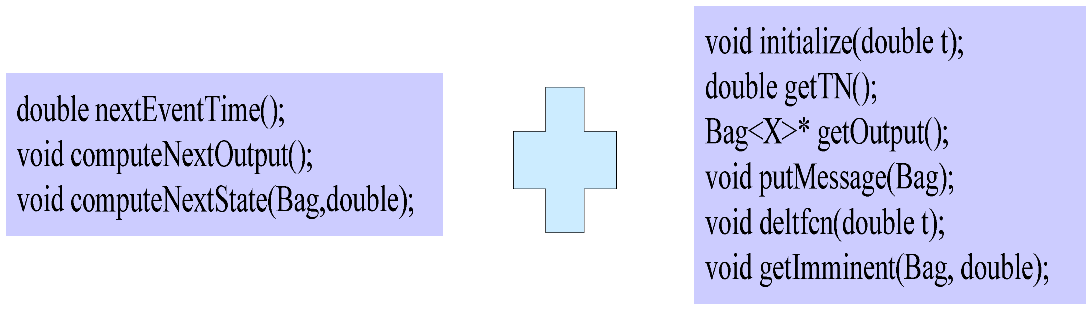
- *nextEventTime*() calls a scheduler's *minPriority*() calculating next event time and classifying imminant models which have next event time and a token for internal transition.
- *computeNextOutput*() executes a *lamda* function if any model has next event time and routes the output message to destination models which have a token for external transition.

- *computeNextState*() executes a delta function according to the token which the models have and initializes the tokens and input/output message containers of models.

The container API consists of a *bag* class, a set class, and an *object_pool* class. The *bag* class holds any type of object and is used in the atomic model and the simulator to store the input or output messages. The *set* class adds some operations with a *set* class which comes from STL. The *object_pool* class is a utility class to handle pools of objects that are stored on the heap and uses the new and delete operators to create and destroy objects.

4.3.2. ADEVS Interface

ADEVS API can not be used directly to create a DEVS simulator service because ADEVS does not provide functions that a DEVS simulator service requires. For example, an method to get time advance (*TA*) is not provided by the ADEVS simulator class. But there is no modification required in the modeling because the ADEVS simulator is used for atomic or coupled model simulation. However, to substitute the functions of ADEVS simulator to the functions of DEVS simulator service in the simulation, some methods are added in the ADEVS simulator as seen in the right side of figure 4-21. The *initialize* function is for initializing an ADEVS model, *getTN* function returns next event time, *getOutput* function returns output message bag from imminent models, *putMessage* function sends the output messages to the corresponding models as input messages, *deltfcn* function lets the models execute their *delt* function, and *getImminent* function is for getting imminent models of all models at the specific time. Those functions are integrated in the ADEVS simulator as seen on the left side of figure 4-21.



```
double nextEventTime();
void computeNextOutput();
void computeNextState(Bag,double);
```

```
void initialize(double t);
double getTN();
Bag<X>* getOutput();
void putMessage(Bag);
void deltfcn(double t);
void getImminent(Bag, double);
```

Figure 4-21 The added functions in the ADEVS simulator for DEVS simulator service

A *String* converter is required due to using C++ standard API and Visual C++ API. Visual C++ is used to create a DEVS simulator web service while C++ standard API is based on the ADEVS. When returning a string to a user in the web service, it should be String class provided in the Visual C++ API. Because a string coming from ADEVS models is C++ standard API, we need to have a string converter of a string for C++ to a String class for Visual C++ and vice versa in the ADEVS interface.

There are two types of classes in the ADEVS for the message passing. One is *PortValue* class and the other is Event class. The *PortValue* consists of a value and a type. The value which is used as a message can be expressed by any class, and the type represents the port which has integer as a default value. The *Event* class is made of a model and a value. The model is a pointer of *Devs<X>* class and the value is a *PortValue*. It is used to route the message to the destination, and the *PortValue* is used for component coupling.

To send the message to another DEVS simulator service from the DEVS simulator service for ADEVS, the message converter is required as the case of simulator service for DEVSJAVA. An imminent model produces a bag containing *PortValue* objects as an output. In the DEVS simulator service, a *getOutput* operation returns a string containing a XML document format as seen on the right side of the figure 4-22. The message structure of ADEVS is shown on the lift side of the figure 4-22. A *Bag* class can get multiple *PortValue* objects consisting of an output port and the message used in the atomic model. The Bag class is written to a XML document which is the same as that of the DEVS simulator service for DEVSJAVA.

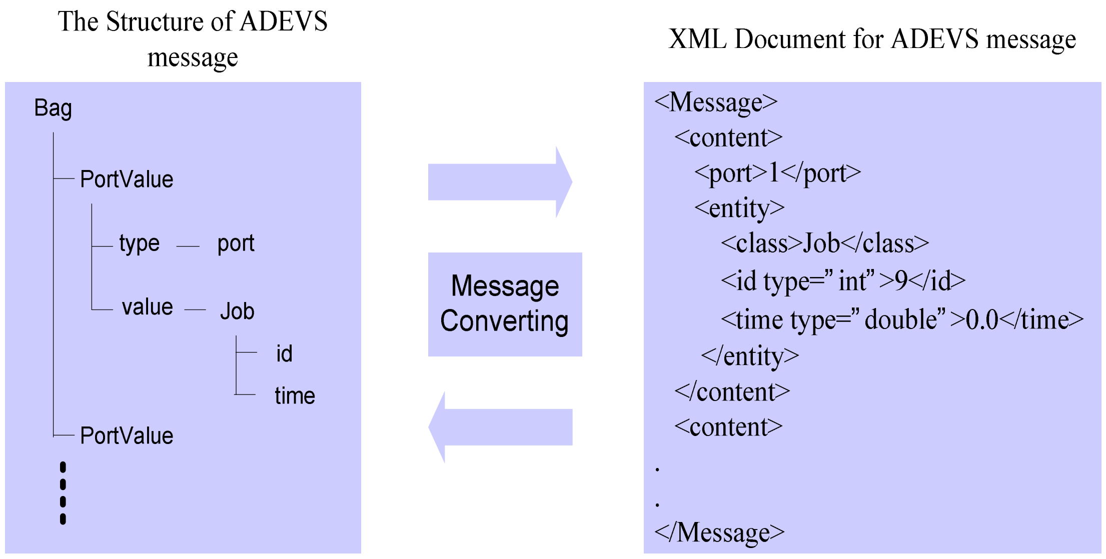


Figure 4-22 The message converting in the DEVS simulator service for ADEVS

To get an XML message from a Bag object, an algorithm for Bag object to XML conversion is used as seen in figure 4-23. In line 1, an instance of *XmlTextWriter* class, which is provided in the visual c++ 2005, is created, and a tag name called “Message” is written using a *WriteStartElement* function in line 2. In line 3, an output Bag is obtained from a *getOutput* function in the *simulator* class. *PortValue* objects are got out from the Bag in line 4. Inside of *while* statement, “content” tag is written and “port” tag is written with a port value using a *WriteString* function in lines 5 to 8. After getting a message of the *PortValue*, “entity” and “class” tags are written and the “class” tag has a name of the instance of the message as a string in lines 10 to 13. Lines 15 to 17 represent a process to make *variable* tags from the message. The *variable* tag has an attribute called “type” and a string value for the variable. The process is repeated in case of multiple variables in the message. Line 19 returns an XML document for the ADEVS message.

```
1.  Make an instance of XmlTextWriter as writer
2.  writer->WriteStartElement(“Message”)
3.  Get an output bag with getOutput() in the simulator class
4.  while(output.size != 0)
5.    writer->WriteStartElement(“content”)
6.    int port = PortValue.port
7.    writer->WriteStartElement(“port”)
8.    writer->WriteString(port)
9.
10.   get an instance of message
11.   writer->WriteStartElement(“entity”)
12.   writer->WriteStartElement(“class”)
13.   writer->WriteString(the name of the instance)
14.
15.   writer->WriteStartElement(the name of the variable)
16.   writer->WriterAttributeString(“type”, the type of the variable)
17.   writer->WriteString(the value of the variable)
18.
19. return the XML document String
```

Figure 4-23 An algorithm for Bag object to XML conversion

The converting of an XML message to an ADEVS message is involves two steps. One is a process to gather information from the XML message using specific class and a list class, and the other is to create an ADEVS message using a *Bag* class containing *Event<X>* objects. Figure 4-24 represents the algorithm for extracting information from the XML message. Line 1 makes the container classes gather information from the XML message and line 2 makes a list for *content* classes which represent “content” tags. The XML document is sent to an *XmlTextReader* class to turn each tag to an object (line 3). The *XmlTextReader* class provides a read function to indicate the end of the document (line 4). If the document reaches end, the *read* function returns *false*. The reader, which is an instance of the *XmlTextReader*, has a *NodeType* to indicate attributes of the nodes. For example, “content” tag is represented as *XmlNodeType::Element*, and if the tag has a text value, the text value is represented as *XmlNodeType::Text*. The closing of “content” tag is represented as *XmlNodeType::EndElement*.

```
1.  make containers to gather information from a XML document
2.  make a list for content classes
3.  turn the XML document to XmlTextReader
4.  while(reader->Read())
5.    switch(reader->NodeType)
6.      case XmlNodeType::Element
7.        get the name from the node
8.        if(isAttributes)
9.          make a pair for name and type and store it into a list for attributes
10.     case XmlNodeType::Text
11.         make a pair for name and value and store it into a list for textvalue
12.     case XmlNodeType::EndElement
13.         if(name is equal to “content”)
14.           make a content instance
15.           while the number of elements in the textvalue is not empty
16.             get name and value from each element of the textvalue
17.             if(name is equal to “port”) assign the value to port variable in the content
18.             elseif(name is equal to “class”) assign the value to class variable in the content
19.             else
20.               make an instance of a threeValue class with name, value, and type variables
21.               assign name and value to the name and the value variables
22.               while number of elements in the attributes is not empty
23.                 get n and t from each element of the attributes
24.                 if name is equal to n
25.                   assign t to the type variables
26.                   put the threeValue into the list on the content class
```

Figure 4-24 The algorithm for extracting information from the XML message

Names of tags are obtained from the *XmlNodeType::Element*. If any tag has an attribute, a pair having a name of a tag and a value of an attribute is stored in the list for attributes (lines 6 to 9). *Textvalues* of tags are provided from the *XmlNodeType::Text*, and a pair with a name of tag and a *textvalue* is saved in the list for *textvalue* (lines 10 to 11). With the *XmlNodeType::EndElement*, the list of content classes is generated with the list for attributes and the list for *textvalues.* If *XmlNodeType::EndElement* is a content tag, an instance of a content class is created, and the list for *textvalue* is used to fill out the content class. A name in a *textvalue* pair is used to select to use a variable of a *ThreeValue* class in the content. If the name is equal to "port", the value corresponding to the name is assigned to a port variable in the content (line 17). If the name is equal to "class", the value is assigned to a class variable in the content (line 18). If the name is not equal to either "port" or "class", the name is considered to be a variable name of a message. In this case, an instance of a *ThreeValue* class is created to accommodate the name, the type and the value of the variable used in the message. The procedure of assigning the name, the type and the value to variables in the *ThreeValue* class is displayed in lines 20 to 25. The variables of the message could be multiple, so the content class has a list for them (line 26).

Once we get a list of content classes, we can create an instance of a *Bag<Event<X>>* class based on information of content classes. Figure 4-25 represents the algorithm to create a Bag instance with *Event<PortValue>* classes. In line 1 a *Bag* instance for *Event* classes is created. A *content* class from a *Bag* is used to generate an *Event* class. The name and the port are obtained from the content class (line 3) and an Event class is created (line

4). The instance of an ADEVS model is assigned to a model variable in the *Event* class, and the port is assigned to a port variable in the value variable (lines 5 to 6). An instance of a class with the name from the content class is created in line 7. The instance is set with values from *threeValue* classes in lines 8 to 9. The instance is assigned to the value for the value variable in the *Event* class in line 10. The *Event* class is put into the *Bag* used to input bag in the simulation.

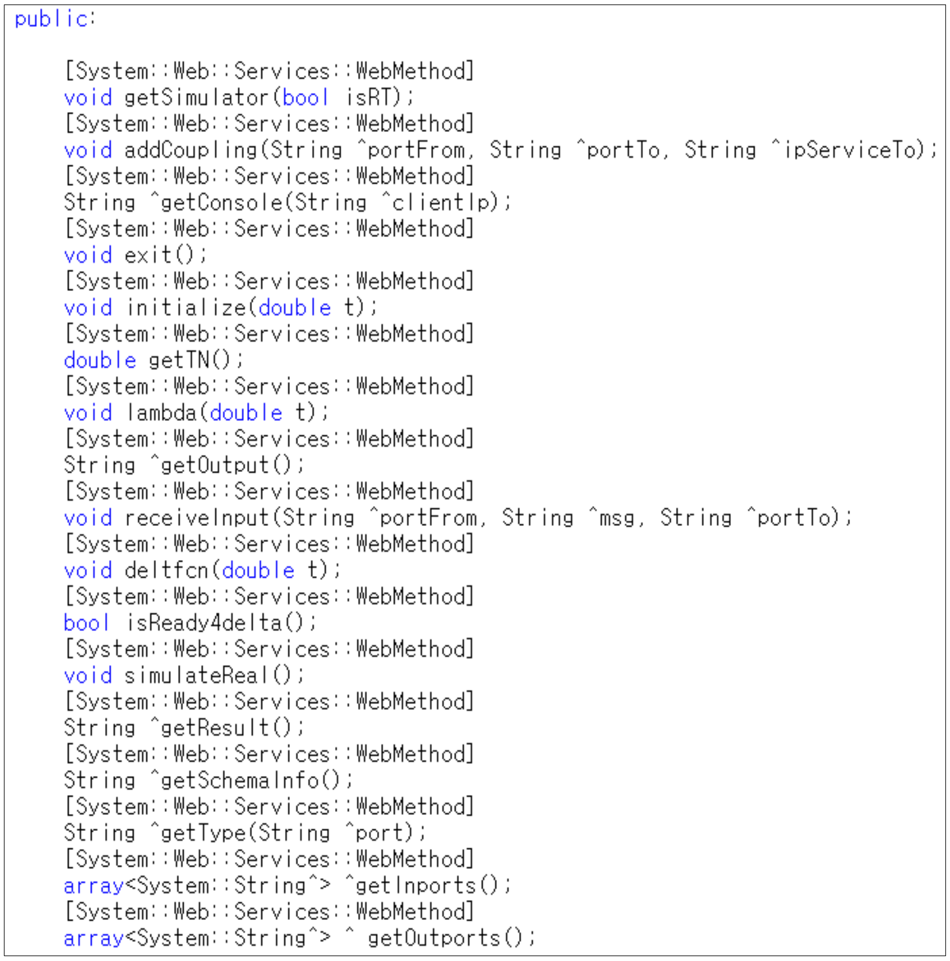
```
 1. make inputs bag for Event<PortValue>
 2. while the list for contents classes is not empty
 3.   get port and name from the content class
 4.   make adevs::Event<PortValue> class with model and value variables
 5.   assign the instance of adevs model to the model variable
 6.   assign the port to the port for the value variable
 7.   create an instance of a class with name
 8.   while the list for threeValue classes is not empty
 9.     set the variables of the instance using the name and the value of the threeValue
10.   assign the instance to the value for the value variable in the Event class
11.   put the Event class into the inputs bag
```

Figure 4-25 The algorithm for creating a *Bag* instance with *Event <PortValue>*

4.3.3. DEVS simulator service with ADEVS

To make web service for ADEVS, visual studio VC++ is used in the .NET environment. The web server is provided by windows OS such as Windows XP or Windows Vista. Visual Studio has a template to create ASP.NET web services [12]. With the template, the operations of the interoperable simulator service are declared, as seen in figure 4-26. Figure 4-26 is a snippet of a header file of the DEVS simulator service, and annotations are used to indicate that methods in the class are used as operations in the web service. *[System::Web::Services::WebMethod]* is an annotation used in the operation.

The web service class inherits a *System::Web::Services::-WebService* class and the name of the web service class becomes the service name.



```
public:

    [System::Web::Services::WebMethod]
    void getSimulator(bool isRT);
    [System::Web::Services::WebMethod]
    void addCoupling(String ^portFrom, String ^portTo, String ^ipServiceTo);
    [System::Web::Services::WebMethod]
    String ^getConsole(String ^clientIp);
    [System::Web::Services::WebMethod]
    void exit();
    [System::Web::Services::WebMethod]
    void initialize(double t);
    [System::Web::Services::WebMethod]
    double getTN();
    [System::Web::Services::WebMethod]
    void lambda(double t);
    [System::Web::Services::WebMethod]
    String ^getOutput();
    [System::Web::Services::WebMethod]
    void receiveInput(String ^portFrom, String ^msg, String ^portTo);
    [System::Web::Services::WebMethod]
    void deltfcn(double t);
    [System::Web::Services::WebMethod]
    bool isReady4delta();
    [System::Web::Services::WebMethod]
    void simulateReal();
    [System::Web::Services::WebMethod]
    String ^getResult();
    [System::Web::Services::WebMethod]
    String ^getSchemaInfo();
    [System::Web::Services::WebMethod]
    String ^getType(String ^port);
    [System::Web::Services::WebMethod]
    array<System::String^> ^getInports();
    [System::Web::Services::WebMethod]
    array<System::String^> ^ getOutports();
```

Figure 4-26 The operations of DEVS simulator service for ADEVS

A web service class is implemented with ADEVS interface and ADEVS modeling and simulation. After finishing writing, the web service class and the project file built in the visual studio, the web service can be automatically deployed to the web server. What is deployed is a bin folder containing a *.dll* file for the web service class, service description file, and web configuration file.

4.4. DEVS Simulator Web Services Integration and Execution

To demonstrate the DEVS simulator service interoperability system, an example DEVS model called *GPT* is used, as seen in figure 4-23. The *GPT* model consists of a coupled model called an Experiment Frame (*EF*) model and an atomic model called a Processer model. The *EF* model has two atomic models called Generator and Transducer. The *GPT* model uses a *Job* type message which has two variables, that is, *id* and *time*. The *Generator* creates new *Job* type messages repeatedly according to the internal time of the model. The *Processer* model processes the *Job* coming from the "in" input port. If the *Job* is finished, it is sent to the *Transducer* which collects information of generated or processed *Job*s and takes the statistics during a certain time. If the certain time is passed, the *Transducer* sends the message to the *Generator* to stop generating a *Job* message.

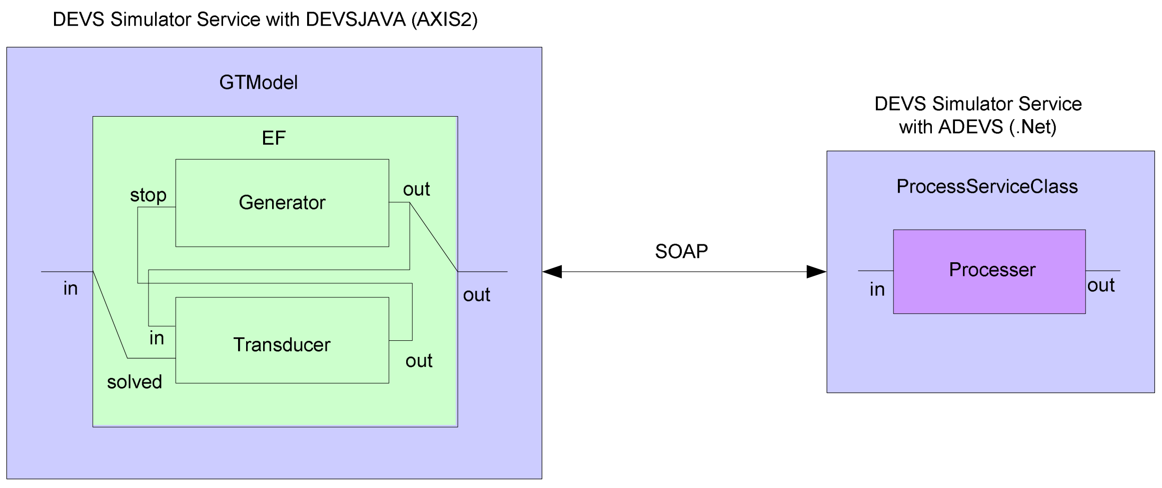
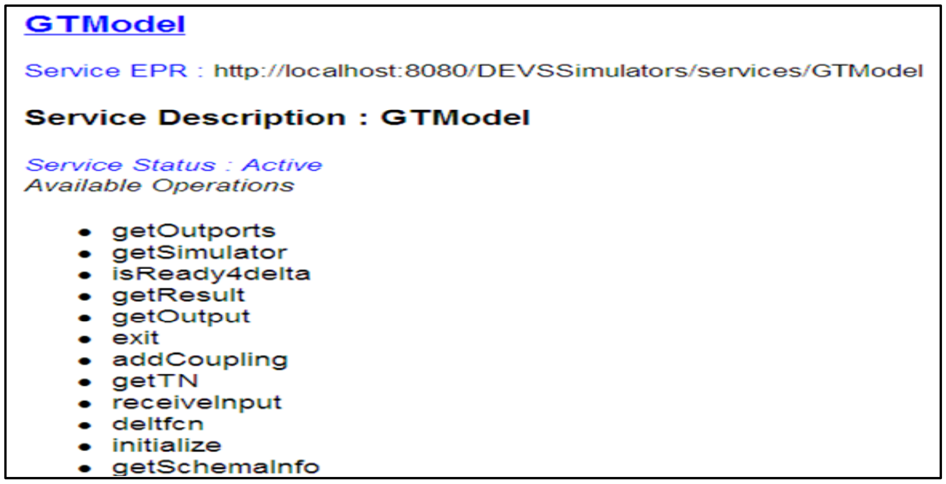


Figure 4-27. The view of the GPT model

Two web services are generated to simulate the GPT model with the interoperable system. One is created with a JAVA based system and DEVSJAVA. Its web service name is *GTModel* containing the *EF* model, as seen in figure 4-28, which displays a view of web service through web browser.



GTModel

Service EPR : http://localhost:8080/DEVSSimulators/services/GTModel

Service Description : GTModel

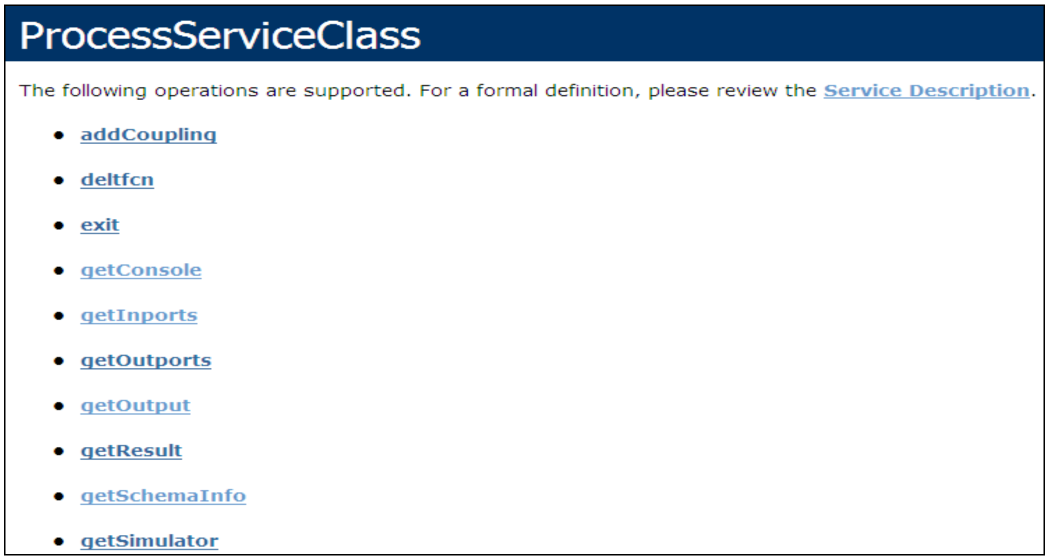
Service Status : Active
Available Operations

- getOutports
- getSimulator
- isReady4delta
- getResult
- getOutput
- exit
- addCoupling
- getTN
- receiveInput
- deltfcn
- initialize
- getSchemaInfo

Figure 4-28 *GT* simulation web service using AXIS2 and DEVSJAVA

The other web service is generated with an ASP.NET based system with VC++ 2005 and ADEVS. *ProcessServiceClass* is the name of the web service embedding the *Processer* model, as seen in figure 4-29, which displays basic information of the service and names of operations used in the web service.

Before a producer of the *EF* web service publishes the *EF* web service, the producer needs to check and register a schema for a DEVS messages used in the *EF* model. In this example, the producer uses a domain name as “EFP”, and *Job* type is used as a DEVS message type, as seen in the figure 4-3. A provider of the processor web service gets a schema for the *EFP* domain from the DEVS namespace and creates the processor web service with .NET and ADEVS. The two web services have common data types for DEVS messages.



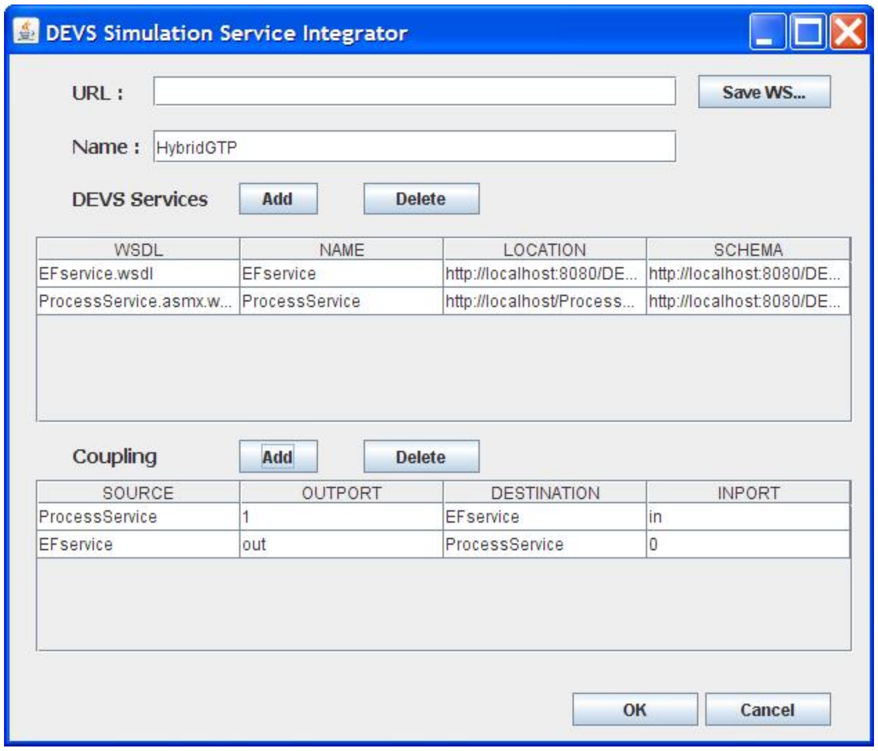
ProcessServiceClass

The following operations are supported. For a formal definition, please review the Service Description.

- addCoupling
- deltfcn
- exit
- getConsole
- getInports
- getOutports
- getOutput
- getResult
- getSchemaInfo
- getSimulator

Figure 4-29 Process model simulator web service using .Net and ADEVS

The DEVS web service integrator is used to integrate two different web services as seen in figure 4-30. The name of integration is “HybridGTP” used for a file name of a XML document. A user can choose a web service, as shown in figure 4-31, appearing when the user clicks an ADD button. The GUI for the information of services displays the information of WSDL selected by the user and sends it to the integrator. In this example, *EFservice.wsdl* and *ProcessService.asmx.WSDL* are selected and their information is shown in the table. Figure 4-32 shows how to couple the web services. The GUI for coupling invokes operations of web services to get their port names and data types of their ports. When finishing the integration, the integrator creates a XML document which contains information of the location of the web service and coupling of selected web services. Figure 4-33 shows the XML document of integration of EF and Processer web services.



DEVS Simulation Service Integrator

URL :

Name : HybridGTP

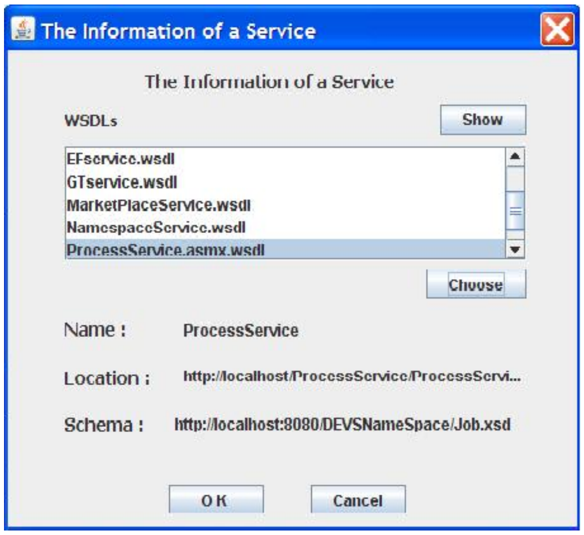
DEVS Services

WSDL	NAME	LOCATION	SCHEMA
EFservice.wsdl	EFservice	http://localhost:8080/DE...	http://localhost:8080/DE...
ProcessService.asmx.w...	ProcessService	http://localhost/Process...	http://localhost:8080/DE...

Coupling

SOURCE	OUTPORT	DESTINATION	INPORT
ProcessService	1	EFservice	in
EFservice	out	ProcessService	0

Figure 4-30. The integrator for EFP web services



The Information of a Service

The Information of a Service

WSDLs

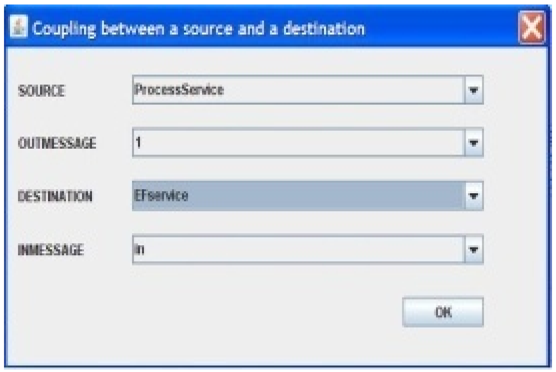
EFservice.wsdl
GTservice.wsdl
MarketPlaceService.wsdl
NamespaceService.wsdl
ProcessService.asmx.wsdl

Name : ProcessService

Location : http://localhost/ProcessService/ProcessServi...

Schema : http://localhost:8080/DEVSNameSpace/Job.xsd

Figure 4-31. The GUI for the information of services


Coupling between a source and a destination

SOURCE: ProcessService

OUTMESSAGE: 1

DESTINATION: EFservice

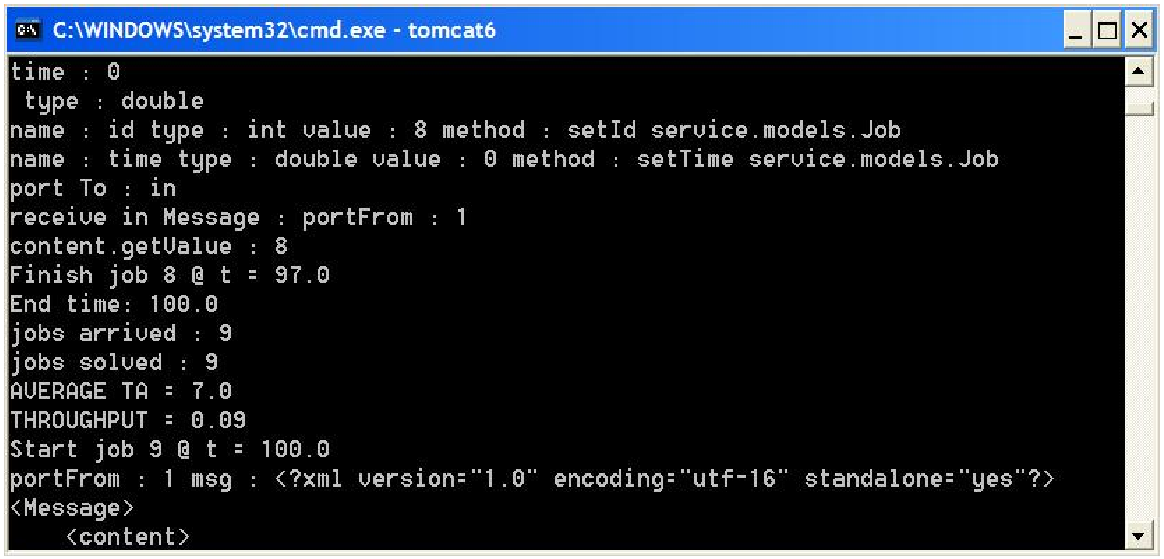
INMESSAGE: in

Figure 4-32. The GUI for coupling between the services

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<devswsintegrator xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="devswsintegrator.xsd">
            <title>HybridEFP</title>
            <services>
                        <model>
                                    <wsdl>EF_Service.wsdl</wsdl>
                                    <name>EF_Service</name>
                                    <location>http://150.135.218.206:8080/DEVSSimulators/services/EFService</location>
                                    <schema>http://150.135.218.199:8080/DEVSNameSpace/Job.xsd</schema>
                        </model>
                        <model>
                                    <wsdl>ProcessService.asmx.wsdl</wsdl>
                                    <name>ProcessService</name>
                                    <location>http://150.135.218.199/ProcessService/ProcessService.asmx</location>
                                    <schema>http://http://150.135.218.199:8080/DEVSNameSpace/Job.xsd</schema>
                        </model>
            </services>
            <couplinginfo>
                        <coupling>
                                    <source>EF_Service</source>
                                    <outport>out</outport>
                                    <destination>ProcessService</destination>
                                    <inport>0</inport>
                        </coupling>
                        <coupling>
                                    <source>ProcessService</source>
                                    <outport>1</outport>
                                    <destination>EF_Service</destination>
                                    <inport>in</inport>
                        </coupling>
            </couplinginfo>
            <inports>
            </inports>
            <outports>
            </outports>
</devswsintegrator>
```

Figure 4-33. The XML document for DEVS Simulator WS Integration

To execute the XML document, we need an execution program called a coordinator, which prepares simulation and runs the simulation. The procedure of the simulation follows the centralized virtual time simulation protocol.



```
C:\WINDOWS\system32\cmd.exe - tomcat6
time : 0
 type : double
name : id type : int value : 8 method : setId service.models.Job
name : time type : double value : 0 method : setTime service.models.Job
port To : in
receive in Message : portFrom : 1
content.getValue : 8
Finish job 8 @ t = 97.0
End time: 100.0
jobs arrived : 9
jobs solved : 9
AVERAGE TA = 7.0
THROUGHPUT = 0.09
Start job 9 @ t = 100.0
portFrom : 1 msg : <?xml version="1.0" encoding="utf-16" standalone="yes"?>
<Message>
    <content>
```

Figure 4-34. The result of simulation of DEVS simulator services

The simulation result is shown in the figure 4-34. It is the same as the result of the GPT model simulated in the DEVSJAVA.

CHAPTER 5. APPLICATION OF INTEROPERABILITY OF DEVS SIMULATOR SERVICES

In this chapter, several applications are introduced with DEVS simulator services and DEVS namespace. The applications need to interoperate with other applications in different platforms and computer languages. The track display and negotiation system are integrated among different language DEVS simulator services implemented in AXIS2 and ADEVS. However, the testing agents system is implemented using DEVSJAVA modeling and simulation, and DEVS simulator services with real time simulator. The testing agents system will show the multiple levels testing concept.

5.1. Track Display

One of the projects, called Automated Test Case Generation (ATCGen) [51], generates DEVS models that are semi-automatically generated from test sequences. A test driver that is based on the DEVS simulation protocol executes the test models in a distributed simulation infrastructure based on the HLA [33]. The test models are DEVS models implemented with C++ language. The integrated system consisting of the test driver and the test model produces its result, which is considered as information of tracks and has a capability of displaying tracks on the track display window.

To display the tracks from DEVS models with heterogeneous computer languages, we should solve interoperability problems between them. The system with DEVS simulator services can communicate between DEVS models with different languages. Accordingly, in this section, we show that the DEVS simulator services and DEVS namespace are applied to solve the interoperable problems with the simple track display system.

5.1.1. Design of Track Display DEVS models

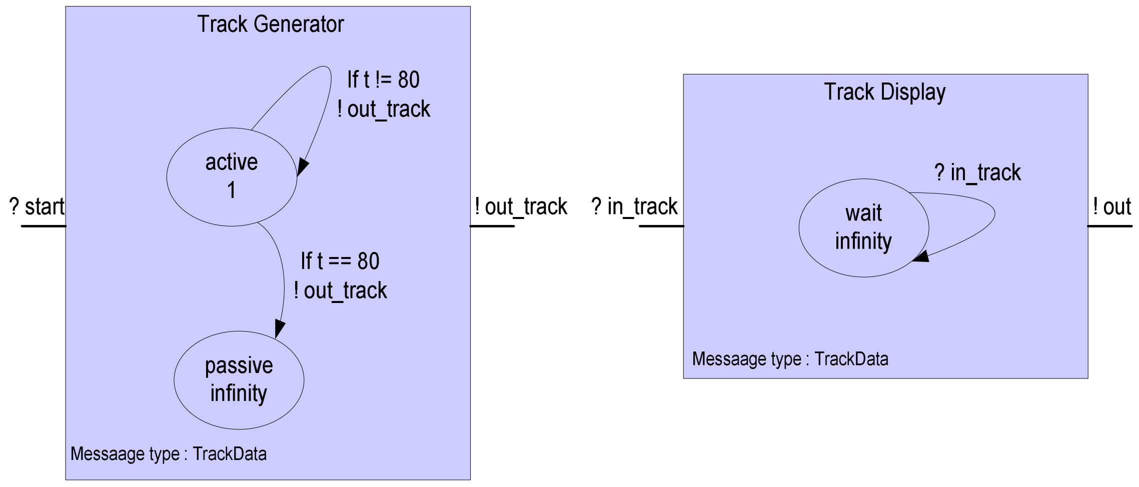


Figure 5-1 State diagrams for track generator and track display

The Track display DEVS model consists of two atomic models called track generator and track display. Figure 5-1 represents state diagrams for the track generator which has two states and two ports, and the track display which has a state and two ports. An initial state of the track generator is an “active” state with 1 unit time, which means the track generator produces an output message and has an internal transition after passing 1 unit time. The output message precedes the internal transition, producing a *TrackData* message. If internal variable t is not equal to 80 units, a next state is the “active” state again with 1 unit time for an internal transition. If t is equal to 80 units, the next state is changed to a “passive” state with infinity units which mean that the track generator does not produce output messages any more.

The track display has a “wait” state with infinity unit time as an initial state. When the track display receives an input message, its state is not changed at all. Instead, a track display window, which resides in the track display model, receives the input message to display the track into the GUI.

The two models share the same message type called a *TrackData*, which has four variables: *id*, *xposition, yposition,* and *heading*. The *id* is integer type and all others are double type. The *id, xposition, yposition,* and *heading* represent a name, longitude, latitude, and direction of a track. To fill out the *TrackData*, the track generator has simple equations to get the longitude, latitude, and heading information. The x value, y value, and heading below (5.1) represent longitude, latitude, and direction of the track, respectively. The velocity and θ are assigned constant values.

$$\begin{aligned} x &= x + velocity * \cos(\theta) \\ y &= y + velocity * \sin(\theta) \\ heading &= \theta \end{aligned} \qquad (5.1)$$

Figure 5-2 shows the view of track display DEVS models using the simView application provided by DEVSJAVA API. There are three DEVS atomic models named “Track Generator”, “Track Generator2”, and “Track Display”. The output port of “Track Generator” and “Track Generator2” are coupled with the input port of “Track Display”.

The initial state of the "Track Generator" and "Track Generator2" is "active" with 1 unit time, while the initial state of the "Track Display" is "wait" with infinity unit time. The *simView* application enables the DEVS model to be simulated with two modes. One mode is to simulate the model with step by step and the other mode is to simulate the model from start to end. The simulation is over when next event time of all models is infinity.

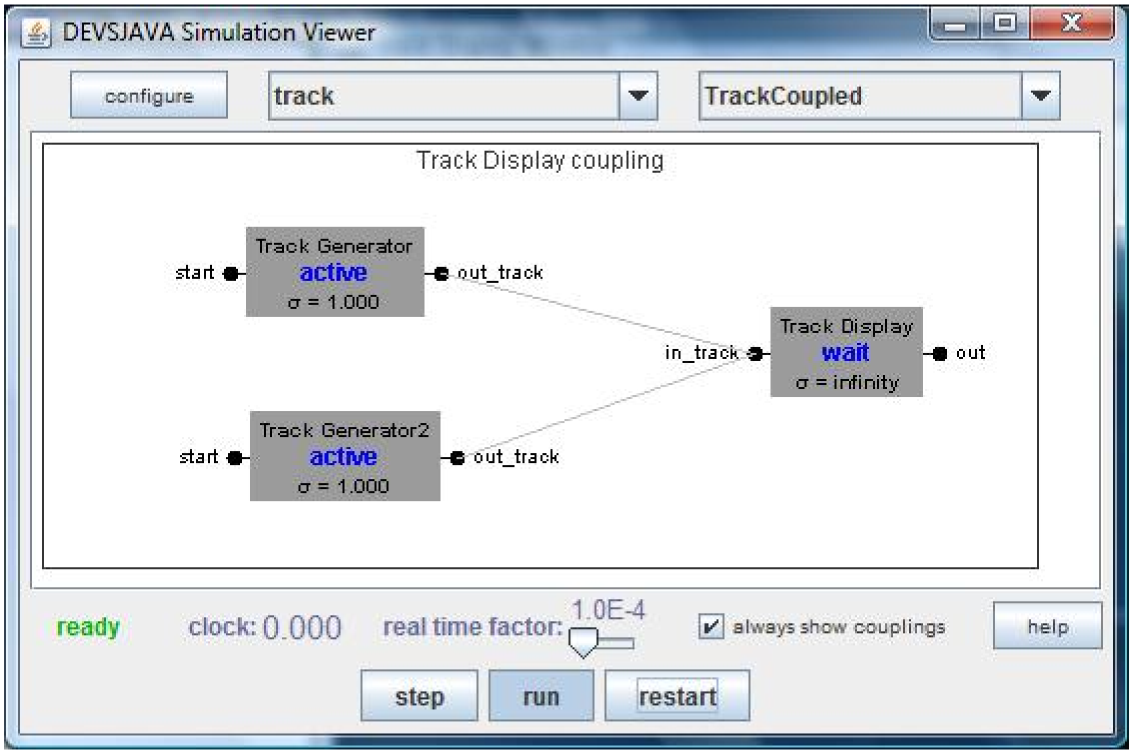


Figure 5-2 The view of Track Display DEVS models with *simView*

5.1.2. Implementation of Track Display with DEVS simulator service

Each model is encapsulated by DEVS simulator service which follows the design concept. The "Track Generator" and "Track Display" reside in *TrackGenerator* and *TrackDisplay* services with DEVSJAVA, AXIS2, and Apache server. The "Track Generator2" model is placed in the *TrackGenerator2* service with ADEVS, .NET, and

Windows server. Before the services are deployed to their servers, data type schema should be registered in the DEVS nameservice with a GUI for schema register. In this case, one message type, called *TrackData*, is used to send the information of the track.

Table 5-1 A message used in the Track Display system

Name of Message	Name of Variable	Type of Variable
TrackData	id	int
	xposition	double
	yposition	double
	heading	doube

Table 5.1 displays the data used in the schema register GUI to generate a schema document for the message. The first column is the name of the message, the second column is the name of the variable, and the third column is the type of the variable. Figure 5-3 represents the schema for the TrackData generated by the schema register GUI.

```
<xsd:complexType name="TrackData">
        <xsd:sequence>
                <xsd:element name="id" type="xsd:int"/>
                <xsd:element name="xposition" type="xsd:double"/>
                <xsd:element name="yposition" type="xsd:double"/>
                <xsd:element name="heading" type="xsd:double"/>
        </xsd:sequence>
</xsd:complexType>
```

Figure 5-3 The schema for the TrackData

We are ready to integrate DEVS simulator services for the Track Display system using the DEVS simulator service integrator. The integrator generates an XML document to describe information of services and coupling information for the track display system.

Figure 5-4 shows the XML document of the track display system. In the document, the locations of services and DEVS namespace and coupling information are displayed.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<devswsintegrator xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="devswsintegrator.xsd">
        <title>TrackDisplaySystem</title>
        <services>
                <model>
                        <wsdl>TrackGenerator.wsdl</wsdl>
                        <name>TrackGenerator</name>
                        <location>http://150.135.218.206:8080/DEVSSimulators/services/TrackGenerator</location>
                        <schema>http://150.135.218.199:8080/DEVSNameSpace/TrackData</schema>
                </model>
                <model>
                        <wsdl>TrackDisplay.wsdl</wsdl>
                        <name>TrackDisplay</name>
                        <location>http://150.135.218.204:8080/DEVSSimulators/services/TrackDisplay</location>
                        <schema>http://150.135.218.199:8080/DEVSNameSpace/TrackData</schema>
                </model>
                <model>
                        <wsdl>TrackGenerator2.asmx.wsdl</wsdl>
                        <name>TrackGenerator2</name>
                        <location>http://150.135.218.199/ProcessService/ProcessTrackGenerator2.asmx</location>
                        <schema>http://150.135.218.199:8080/DEVSNameSpace/TrackData</schema>
                </model>
        </services>
        <couplinginfo>
                <coupling>
                        <source>TrackGenerator</source>
                        <outport>out_track</outport>
                        <destination>TrackDisplay</destination>
                        <inport>in_track</inport>
                </coupling>
                <coupling>
                        <source>TrackGenerator2</source>
                        <outport>out_track</outport>
                        <destination>TrackDisplay</destination>
                        <inport>in_track</inport>
                </coupling>
        </couplinginfo>
        <inports>
        </inports>
        <outports>
        </outports>
</devswsintegrator>
```

Figure 5-4 The view of the XML document for the track display system

According to the XML document, the *TrackGenerator* service is located in the http://150.135.218.206:8080 server, the *TrackDisplay* service is located in the http://150.135.218.204:8080 server, and the *TrackGenerator2* service and DEVS namespace are located in the http://150.135.218.199 server with ports 80 and 8080, respectively. To execute the XML document, the track display window GUI is shown in the server containing the *TrackDisplay* service.

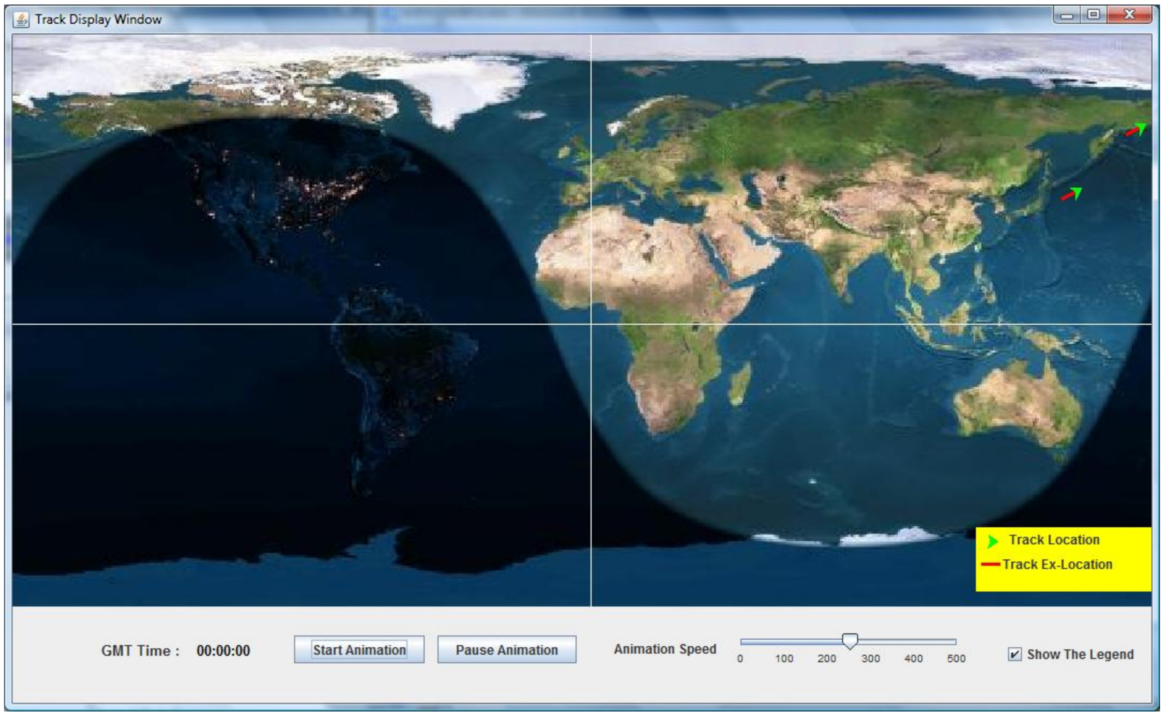


Figure 5-5 The track display window

Figure 5-5 shows the track display window GUI in the *TrackDisplay* DEVS model. The track display window displays tracks with a triangle shape in green on the world map. Two tracks, one from *TrackGenerator* service and the other from TrackGenerator2 service, are shown on the right upper side in the map. The red line shows the trajectory of the track.

Through the track display system consisting of DEVS simulator services, we know that a track display model in DEVS simulator service can have a capability of displaying tracks from the DEVS simulator services if the message between a track generator and a track display is the same even though the services do not have the same platform and language. The track generator can be any system like *ATCGen* to generate the information of tracks based on the simulation.

5.2. Negotiation System

[74] proposed a negotiation system to be used in the different domains by defining the dynamic message structure with the System Entity Structure (SES). Also, [74] suggested an automated domain independent marketplace architecture where user agents can interact with providers with negotiation protocols that describe the policy of communications in multi-agent environments. In the [74], there is an example of distributed services environment called printing jobs scenario where users send different kinds of printing jobs to the marketplace which selects the best providers, and negotiates on different aspects of the job specifications with the providers until they reach an acceptable agreement within their range. The user and the provider, who are represented as DEVS models with specific behaviors, are considered as agents in the negotiation system. The printing jobs with marketplace are implemented with DEVSJAVA to validate the concept proposed in [74]. The behaviors of the user, the providers, and the marketplace are more complicated than the track display system, and the printing jobs system has more messages than the track display system.

The printing jobs system can be extended to accommodate agents who are created in the different platforms and languages using the DEVS simulator services. We use DEVSJAVA models and ADEVS models to create the printing jobs system with an agent from a different environment. The printing jobs system basically has three models called a user, a provider and a marketplace model. One of the providers is modeled with ADEVS and others are modeled with DEVSJAVA. Each model is capsulated in the DEVS simulator service and integrated to make the printing jobs system. Through the simulation of the integrated system, we will show that the negotiation system can be used by agent models from different languages

5.2.1. Design of Negotiation System with DEVS simulator service.

The negotiation system for printing jobs consists of three models called user, provider, and marketplace model which have their own behavior for negotiation. The behavior of the marketplace model is represented in figure 5-6.

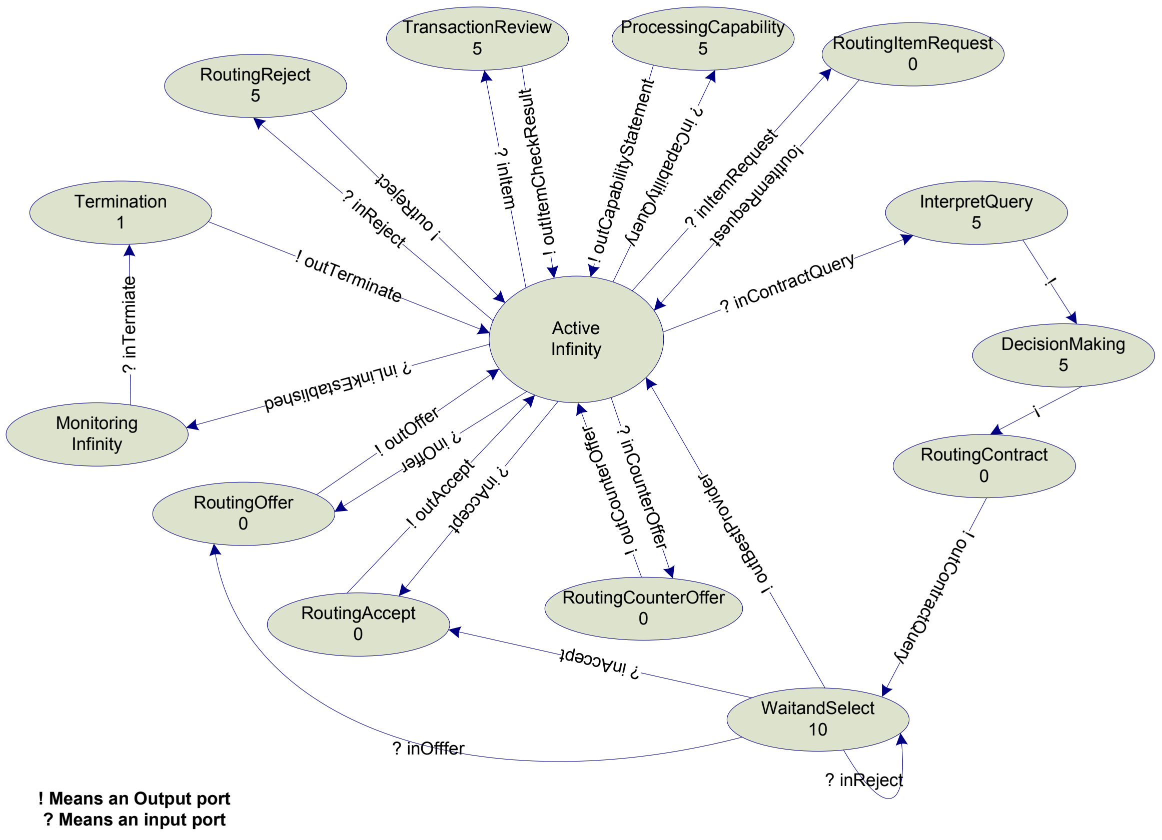


Figure 5-6 The state diagram of the marketplace model

The initial state of the marketplace model is “Active” state with infinity unit time. When the marketplace receives a *CapabilityQuery* message from the user, its state is changed to “ProcessingCapability” with 5 unit time, and comes back to “Active” states after passing 5 unit time and generating a *CapabilityStatement* message. If the *CapabilityStatement* is satisfied, the user sends a *ContractQuery* message to the marketplace. When the *marketplace* receives the message, its state is changed to “InterpretQuery” with 5 units. After 5 units, its state is changed to “DecisionMaking” with 5 units. In this state, the *marketplace* decides the providers to cover the contract with its database for providers. After finishing selecting the providers, the *marketplace* routes the contract to the selected providers in the “RoutingContract” state. In the “WaitandSelect” state, the marketplace can receive three different types of messages, that is, Reject, Accept, and Offer. When the marketplace receives a Reject message, it stays on the “WaitandSelect” state. If it receives an Accept message, it changes its state to “RoutingAccept” state, produces an Accept message right away, and goes to “Active” state. If it receives an Offer message, its state is changed to “RoutingOffer” with 0 unit, it sends an Offer message to the user, and its state becomes “Active” state. The marketplace has a role to hand the message to the user or the providers. For example, the user provides a “CounterOffer” message to the *marketplace*, and the marketplace sends the message to the providers. Conversely, the provider sends an “Offer” message to the marketplace, and the *marketplace* hands the message to the user. If the user and the provider agree in the negotiation with each other, the marketplace expects a “LinkEstablished” message, it goes to “Monitoring” states with infinity, and the negotiation is terminated after receiving a “Terminate” message from the user and providers.

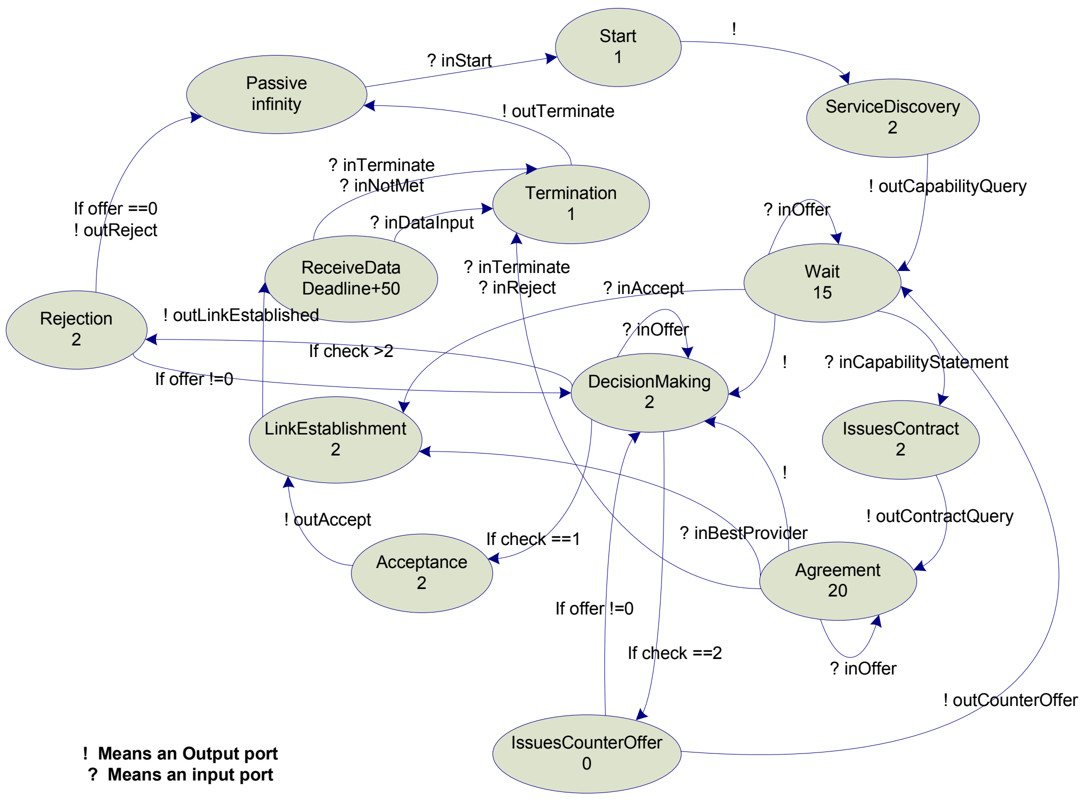


Figure 5-7 The state diagram for the user model

Figure 5-7 represents the state diagram for the user model in the negotiation system for printing jobs service. The initial state of the user model is “Start” state with 1 unit. After passing 1 unit, the user of the printing services changes its state to “ServiceDiscovery” state with 2 units, sends a “CapabilityQuery” message to the marketplace after passing 2 units, and goes to “Wait” state with 15 units. If the user receives the “CapabilityStatement” message and the message satisfies the user, its state is changed to “IssueContract” with 2 units. After 2 units, it changes its state to “Agreement” state with 20 units and sends a ContractQuery message to the marketplace. During the “Agreement” state, it can receive four messages called a *BestProvider*, an *Offer*, a

Terminate, and a *Reject*. The *BestProvider* changes the state of the user to "LinkEstablishment" state with 2 units, the *Offer* makes the state of the user stay on "Agreement" state with the next event time, and the *Terminate* and the *Reject* change the user's state to "Terminate" state with 1 unit. When the user spends its internal event time, its state is changed to "DecisionMaking" state with 2 units. In the "DecisionMaking" state, the user evaluates the offers from marketplace with its own decision processing, and produces an integer value called "check". According to the value of the "check" variable, the user's state is changed to "Acceptance" with 2 units in check = 1, "IssuesCounterOffer" with 0 unit in check = 2, and "Rejection" with 2 units in check > 2. In case of "IssuesCounterOffer", the user sends a CounterOffer message and changes its state to "Wait" state with 15 units. In "Rejection", if the offers do not exist in the offer bag, the user's state becomes "Passive", which means the negotiation is over. In case of "Acceptance", the user sends an Accept message and its state becomes "LinkEstablishment" state with 2 units after which it sends a LinkEstablished message and its state is changed to "ReceiveData" state with Deadline+50 units. If the user receives a *DataInput* message, its state goes to "Terminate" state with 1 unit. This stage means the negotiation for printing jobs service is successfully finished. If the user receives a Terminate or NotMet message, the negotiation is unsuccessfully terminated.

The user has certain rules to accept the *offer* from the providers. If the acceptance condition is not satisfied, the user sends a *counteroffer* to the providers until the acceptance condition is met.

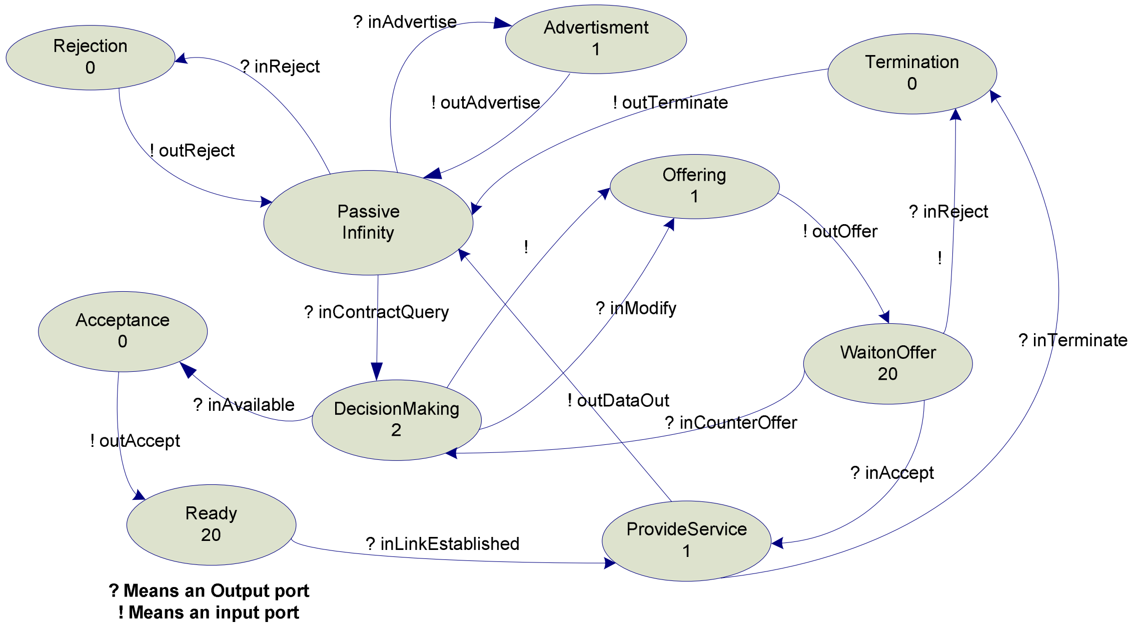


Figure 5-8 The state diagram for the provider model

Figure 5-8 represents the state diagram for the provider model whose initial state is "Passive" with infinity for next internal event time. When the provider receives a *ContractQuery* message, its state is changed to "DecisionMaking" with 2 units. In this state, the provider proposes the offer for the contract and changes its state to "Offering" with 1 unit after which it produces an *Offer* message and changes its state to "WaitonOffer" with 20 units. If 20 units are passed, the provider goes to "Termaination" state, sends a Terminate message right away and changes its state to "Passive". If the provider receives a CounterOffer message in the "WaitonOffer", its state is changed to "Decision" with 2 units. If the provider receives an *Accept* message, it changes its state to "ProvideService" with 1 unit after which it produces a DataOut message and changes its state to "Passive". The above processes represent the processes of negotiation in the provider model.

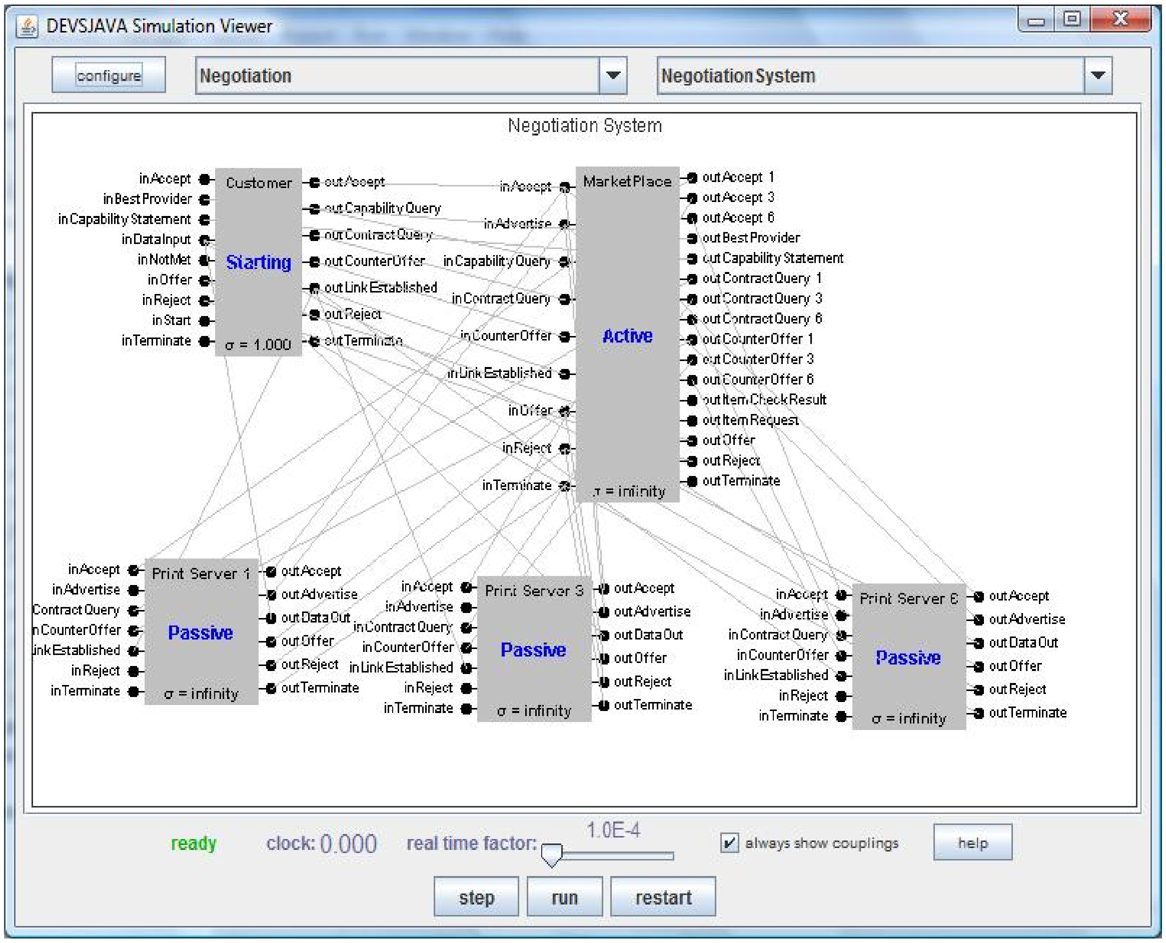


Figure 5-9 The negotiation system model for a printing jobs service

The overall negotiation system is displayed in the figure 5-9 containing five atomic models called *Customer, MarketPlace, Print Server 1, Print Server 3, and Print Server 6*. The *Customer* is the user model *and Print Server 1, 3, and 6* are the provider models. The Customer model has decision making rules as follows [74].

- The Customer model is going to find a provider that has the Business Cards printing capability.
- The Customer would accept an offer if one of the following conditions is

satisfied:

- ■ If the paper quality is medium or high, the color is full HD and the deadline is less than 80.
- ■ If the paper quality is medium or high, the color is RGB and deadline is less than 30.
- ■ If the paper quality is medium or high, the color is grayscale and the deadline is less than 20.

- ● If the offer does not match any of its acceptable ranges, the user sends back a counter offer asking either his first preference or a modified one based on the history of the offers he was receiving.

The *Print Server 1, 3, and 6* have their capability of providing printing services and propose offers based on their own data. In this model, each Print Server has its data in the model and provides a random value to calculate the current *Deadline* for its printing service. For instance, current *Deadline* is the subtraction of its own random value from previous deadline. The marketplace has its own database to answer the capability query from the customer. In this model, the database is confined to the Business Cards service which the customer requests.

The result of the simulation of the negotiation system is following.

```
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
Offer information are:
Customer : Customer
Job Type : Business Cards
Print Server : Print Server 6
Color : FullHDColor
Paper Quality : High
Deadline : 72
```

```
Duplex : Yes
Number of Copies : 1
Technology Type : Thermography
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
```

5.2.2. Implementation of Negotiation System with the DEVS simulator service

Each model is encapsulated in the DEVS simulator service for DEVSJAVA or ADEVS. The *Customer, MarketPlace, Print Server 1, Print Server 3* is placed in the DEVS simulator service for DEVSJAVA, and *Print Server 6,* which is generated with ADEVS, is embedded in the DEVS simulator service for ADEVS. The models have unique ports with specific data type that should be registered to DEVS namespace. The data types used in the negotiation system are displayed in the table 5-2.

Table 5-2 The messages used in the negotiation system

Name of Message	Name of Variable	Type of Variable
CapabilityQuery	customer, printJob	String
CapabilityStatement	printJob, printServer	String
ContractQuery	printJob, technologyType, noCopies, deadline, customer, paperQuality, duplex, printJobID, color	String
CounterOffer	printJob, technologyType, noCopies, deadline, customer, paperQuality, duplex, printJobID, color, printServer	String
Offer	printJob, technologyType, noCopies, deadLine, customer, paperQuality, duplex, printJobID, color, printServer	String
Accept	customer, printServer, printJobID	String

Advertise	provider, content	String
DataOut	user, content	String
LinkEstablished	customer, printServer	String
Reject	customer, printServer, printJobID	String
Terminate	msg	String

Each port in the models should be mapped to the one of messages above. The information of mapping ports to message is added in the DEVS simulator service. Each message is converted to a schema document which is stored in the DEVS namespace. Figure 5-10 represents the schema document for the *ContractQuery* message which consists of nine variables that have string type. Other messages are converted to the schema documents and registered to the DEVS namespace using schema register GUI. In the DEVS simulator service, the information of schema location, input ports array, output ports array, and mapping port to message type are needed to couple between services. Based on the information, the client consuming the services can compose the negotiation system with the DEVS simulation service integrator.

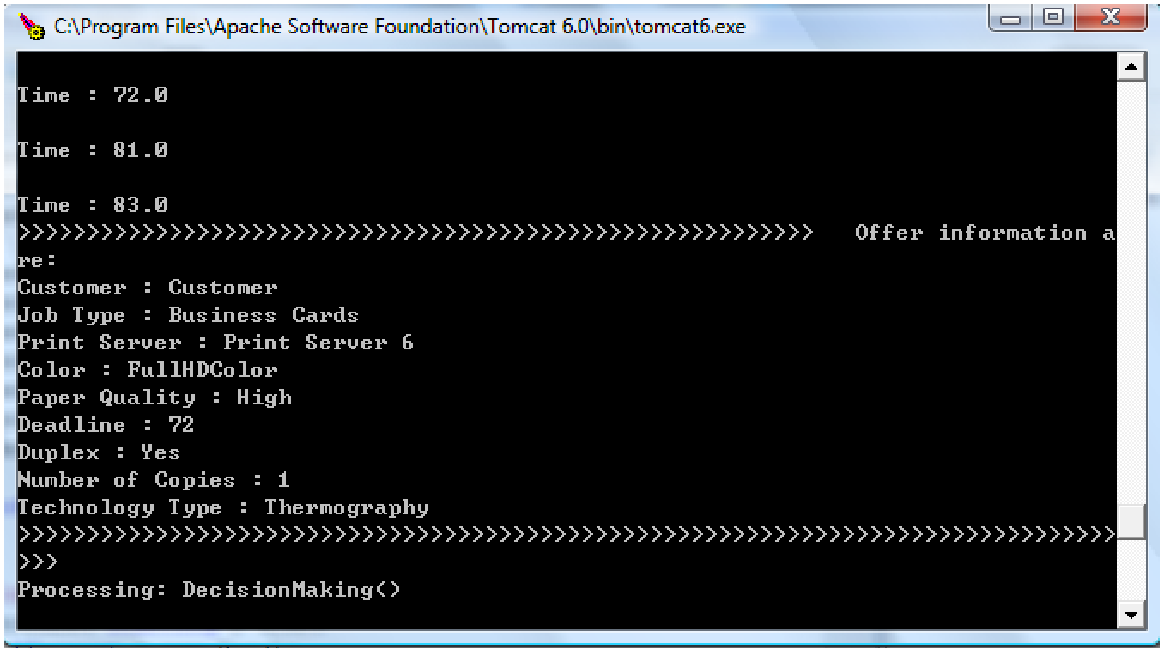
```
<xsd:complexType name="ContractQuery">
        <xsd:sequence>
                <xsd:element name="printJob" type="xsd:string"/>
                <xsd:element name="technologyType" type="xsd:string"/>
                <xsd:element name="noCopies" type="xsd:string"/>
                <xsd:element name="deadline" type="xsd:string"/>
                <xsd:element name="customer" type="xsd:string"/>
                <xsd:element name="paperQuality" type="xsd:string"/>
                <xsd:element name="duplex" type="xsd:string"/>
                <xsd:element name="printJobID" type="xsd:string"/>
                <xsd:element name="color" type="xsd:string"/>
        </xsd:sequence>
</xsd:complexType>
```

Figure 5-10 The schema document for the ContractQuery message

Table 5-3 Assignment of DEVS simulator services to servers

Server name	Services and Client	Method of implementation
150.135.218.199	Print Server 6, DEVS namespace	ADEVS
150.135.218.201	Print Server 1, Print Server 3	DEVSJAVA
150.135.218.204	Customer	DEVSJAVA
150.135.218.206	MarketPlace DEVS service integrator	DEVSJAVA

Table 5-3 shows the assignment of the services to servers and method of implementation. To execute the negotiation system using DEVS simulator services, we use four machines running their server as seen in the first column of table 5-3. Each server has DEVS simulation services shown in the second column of table 5-3. The service in the 150.135.218.199 uses ADEVS and .NET environment, but others use DEVSJAVA and AXIS2.


C:\Program Files\Apache Software Foundation\Tomcat 6.0\bin\tomcat6.exe

```
Time : 72.0

Time : 81.0

Time : 83.0
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   Offer information a
re:
Customer : Customer
Job Type : Business Cards
Print Server : Print Server 6
Color : FullHDColor
Paper Quality : High
Deadline : 72
Duplex : Yes
Number of Copies : 1
Technology Type : Thermography
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>
Processing: DecisionMaking()
```

Figure 5-11 The result of the negotiation system using DEVS simulator services

Figure 5-11 shows the result of the negotiation system using DEVS simulator services. The result of simulation of integrating the services is the same as that of simulation of the DEVSJAVA model. Through the negotiation system, a provider called Print Server 6 implemented with different language (ADEVS), can be simulated with heterogeneous DEVS models (DEVSJAVA) using the DEVS simulator service concept. Fairly complex models with different implementation methods can be interoperated under the DEVS simulator services.

5.3. Test Agents for Net-centric

Test agents for Net-centric have different levels of testing capability of interaction between the user and the provider through the web services. The different levels are divided into three layers called syntactic layer, semantic layer, and pragmatic layer. The syntactic layer belongs to common formats and protocols for communicating message data frames [25]. The semantic layer includes share of meaning of the message between a sender and a receiver. The pragmatic layer employs the shared agreements about the use of information exchanged. For example, the receiver reacts to the message in a manner that the sender intends [25]. With test agents, a system using web services to corroborate among participants can be simultaneously tested at the multiple layers.

This test agents system has two sub-systems where one is participant models with DEVS agents, and the other is observer models watching the participant models to verify the participant’s behaviors. The participant models interact with each other through the web services, and the observer models are distributed in the networks forming the web services. The DEVS agents in the participants send messages containing the information of invocation of web services to the observer services which send the messages to other observer services to notify what the participants do.

In this section, the observer models turn to the DEVS simulator services called observer services. With the integration of the observer services, observing environments are constructed through a message type matching method using DEVS namespace. We will implement the test agents with a modified negotiation system to show the possibility of multiple layer testing.

5.3.1. Design of Test Agents for Net-centric

5.3.1.1. Modified negotiation system

The negotiation system in the previous section consists of DEVS models, but a modified negotiation system includes DEVS models and a marketplace web service. The marketplace model is substituted to the web service in the modified negotiation system to make an environment of collaboration between a user model and a provider model. The user model and the provider model need to be changed in their modeling to communicate with the marketplace web service.

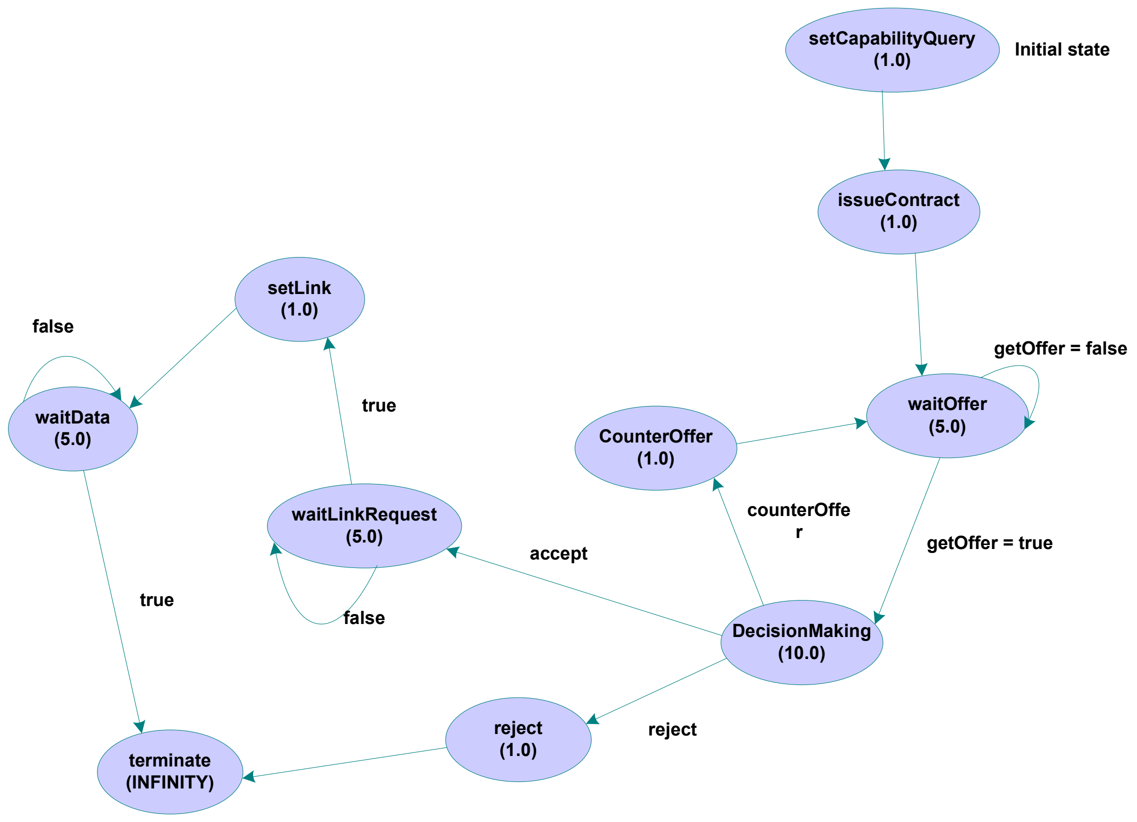


Figure 5-12 The state diagram for a modified user model

The state diagram for a modified user model is displayed in figure 5-12 which shows the initial state of the user as “setCapabilityQuery” state with 1 unit. In this state, the user model invokes a web service to send capabilityQuery to the marketplace service, and it receives the result of the capabilityQuery. If the result from the marketplace service satisfies the user, the user produces a contract for the marketplace service through invoking a web service, and it changes its state to “waitOffer” with 5 units. In the “waitOffer” state, the user is waiting for an offer by invoking a web service for getting an offer. If a getOffer variable is false after getting a result of invocation of the web service, the user stays on the “waitOffer” state. While the getOffer is true, the user changes its state to “DecisionMaking” state with 10 units. The user decides on the result for the offer, and if the decision is a counteroffer, its state is changed to “CounterOffer” with 1 unit, after which it invokes the web service for sending the counteroffer message. If it is rejected, the user alters its state to “reject” with 1 unit, after which its state is changed to “terminate” with infinity units. If accepted, the user changes its state to “waitLinkRequest” with 5, after which if a result of invocation of a web service is false, its state stays on the same state, but if true, the user establishes the link with a provider. At that time, its state is “setLink” state with 1 unit. After passing 1 unit, the user waits to receive data from a provider with “waitData” state. If the user receives the data, the negotiation is successfully over, and the user has “terminate” state with infinity.

Figure 5-13 shows the state diagram for a provider model whose initial state is “waitContract” with 5 units after which, if a result of invocation of a web service is false, the provider is still in the “waitContract” state. If true, it changes its state to “makeDecision” with 10 units after which it changes its state to “offer” with 1 unit. Passing 1 unit, the provider invokes a web service for sending an offer to the user, and its state alters its state to “waitReply” with 5 units, after which, if the provider gets the reply from the user, it changes its state to “makeDecision”, “setRequestLink”, and “EndNego” according to the return values called *counteroffer, accept, and reject*, respectively. If not, the provider stays on “waitReply”. In case of counteroffer, the states make a loop until the provider receives the *accept* result. In case of *reject*, the provider stops the negotiation with the user staying in “EndNego”. If the provider receives *accept* from the user, it changes its state to “setRequestLink” state with 1 unit, after which it invokes a web service to request link to the user and goes into “waitLink” state with 5 units. In “waitLink”, if the result from the web service is *false*, the provider’s state stays in “waitLink”, but if *true*, its state is altered to “setData”, and it invokes the web service to send *Data* to the user. After that, the provider finishes the negotiation with the user with “EndNego” state.

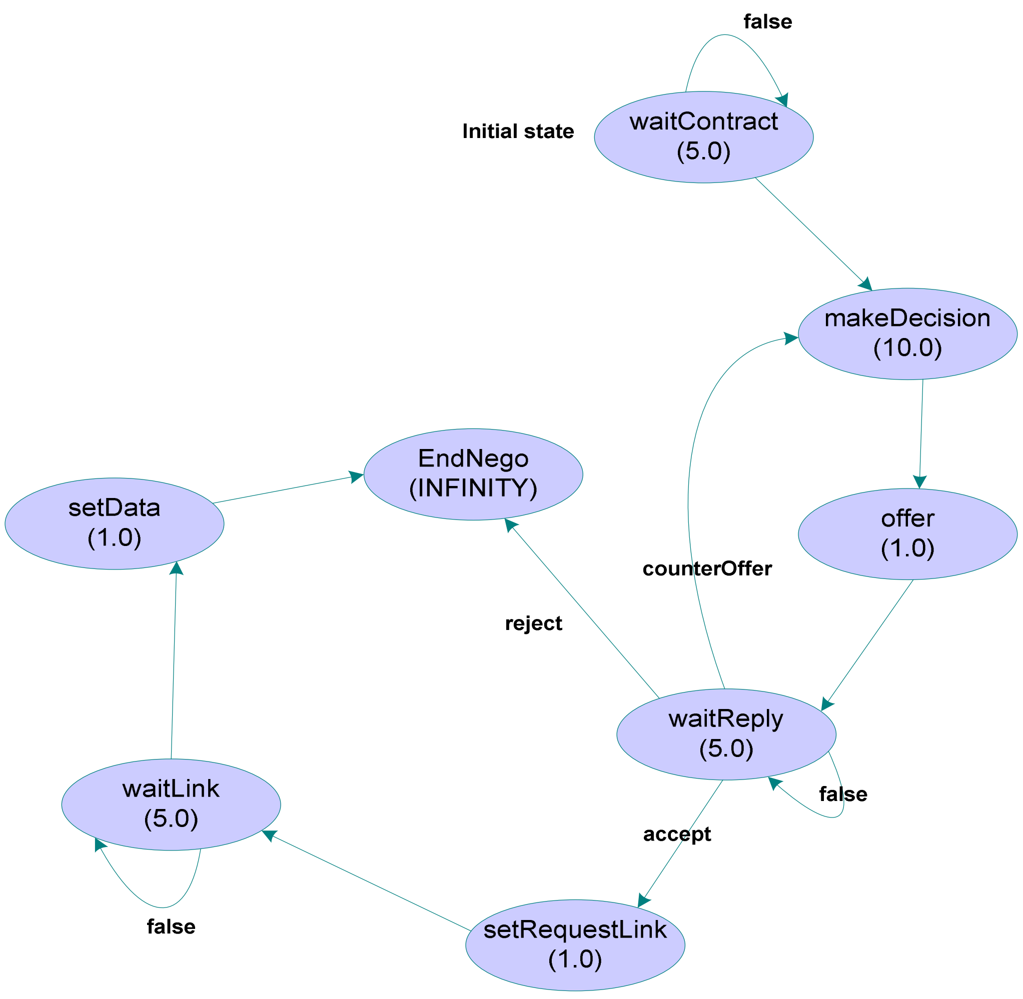
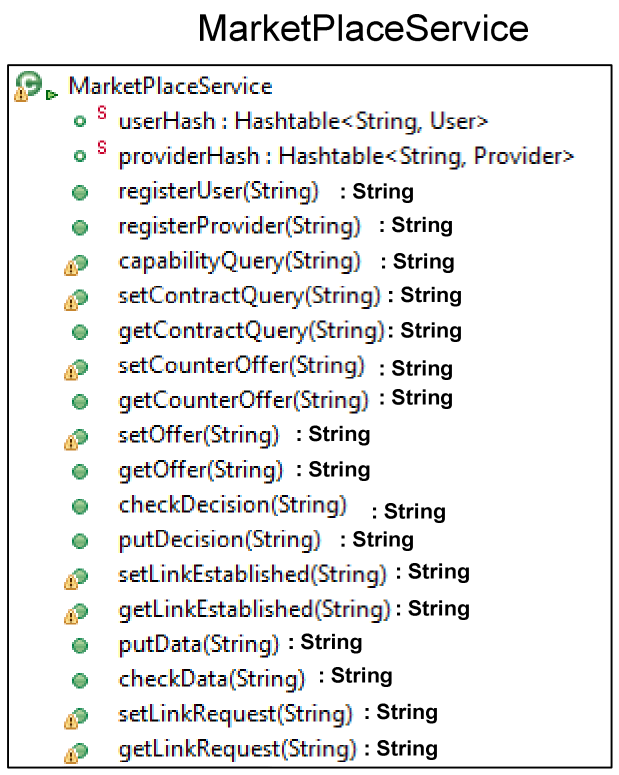


Figure 5-13 The state diagram for a modified provider model



MarketPlaceService

```
MarketPlaceService
    S userHash : Hashtable<String, User>
    S providerHash : Hashtable<String, Provider>
    registerUser(String)   : String
    registerProvider(String)   : String
    capabilityQuery(String)   : String
    setContractQuery(String) : String
    getContractQuery(String): String
    setCounterOffer(String)   : String
    getCounterOffer(String)  : String
    setOffer(String)   : String
    getOffer(String)   : String
    checkDecision(String)   : String
    putDecision(String)   : String
    setLinkEstablished(String) : String
    getLinkEstablished(String) : String
    putData(String) : String
    checkData(String)  : String
    setLinkRequest(String)  : String
    getLinkRequest(String) : String
```

Figure 5-14 The operations of marketplace service

Figure 5-14 represents the operations of the marketplace service which has 17 operations having a string argument and a string return value, and two containers for a user and a provider. When a user and a provider models are initialized, they register their names in the *hashtable* to make instances of user and provider class to put the messages in the instances. The *capabilityQuery* operation is for searching a provider which has capability that a user requests. The *setContractQuery* is for putting the contract into the instance of the provider and the *getContractQuery* is for picking up the contract from the instance of the provider. The *setCounterOffer* is to set a counteroffer into the instance of the provider, and the *getConuterOffer* is to get the counteroffer from the instance. The *setOffer* and *getOffer* are used to send an offer to the user. The *checkDecision* and the *putDecision* are used to let their opposite know their decision. The *setLinkEstablished* and the *getLinkEstablished* are used to send information for link from the user to the provider. The *putData* and *checkData* are for sending data from the provider to the user. The *setLinkRequest* and *getLinkRequest* are for requesting link between the user and the provider.

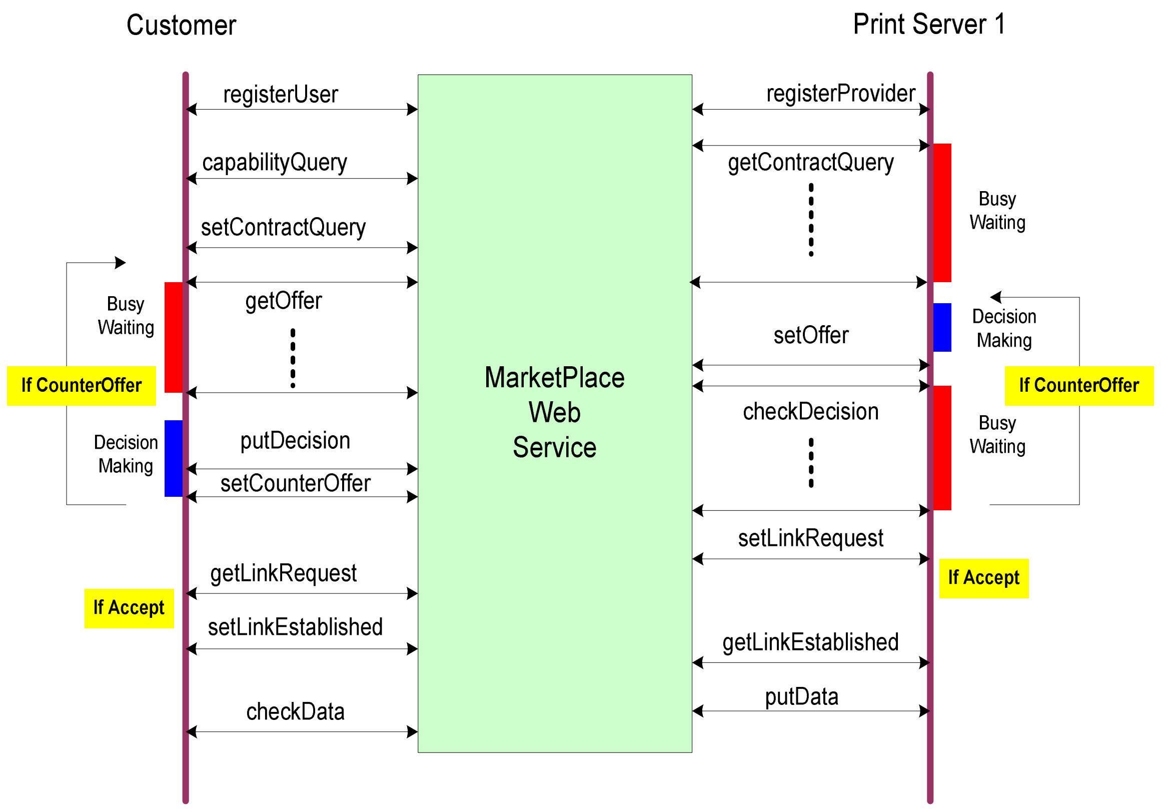


Figure 5-15 Interaction between a user and a provider through marketplace web service

Figure 5-15 represents the interaction between a user and a provider through a marketplace web service. The Customer is an instance of a user, and the Print Server 1 is an instance of a provider. The Customer and the Print Server 1 use the marketplace web service to communicate with each other. Bidirectional arrows between the Customer and the MarketPlace web service, and between the MarketPlace web service and the Print

Server 1, represent invocation of operations. The Customer invokes nine operations in the web service, while the Print Server 1 invokes seven operations. Red rectangles mean busy waiting whose purpose is to get the information from the server. Blue rectangles mean a period for decision making after which the Customer generates the message regarding the decision making. If the decision making is a counteroffer, the Customer is moved to a position to get the offer from the Print Server 1, and after the Print Server 1 gets the information for the Customer's decision, it is moved to a position of decision making. If accepted, the two models successfully finish their negotiation.

5.3.1.2. Observer models

There are two observer models called a user observer and a provider observer. Figure 5-16 represents the state diagram for the user observer whose initial state is "waitForCapabilityQuery" with infinity units. There are rules to make ports according to the sources. For example, "waitForCapabilityQuery" state waits for an input message in the inCapabilityQuery port. The input port receives a message sent by the Customer model in the negotiation system. If the user observer receives a CapabilityQuery message, it changes its state to "sendOutCapabilityQueryAlert", and produces a Capability-QueryAlert message to the provider observer model. The output port, whose name has "Alert", connects to the input port whose name has "Alert". The input port, whose name has "Result", connects to the output port whose name has "Notice". When the observer waits for an offer from the provider observer, its state is "waitForOffer" until it receives an OfferAlert message from the provider observer.

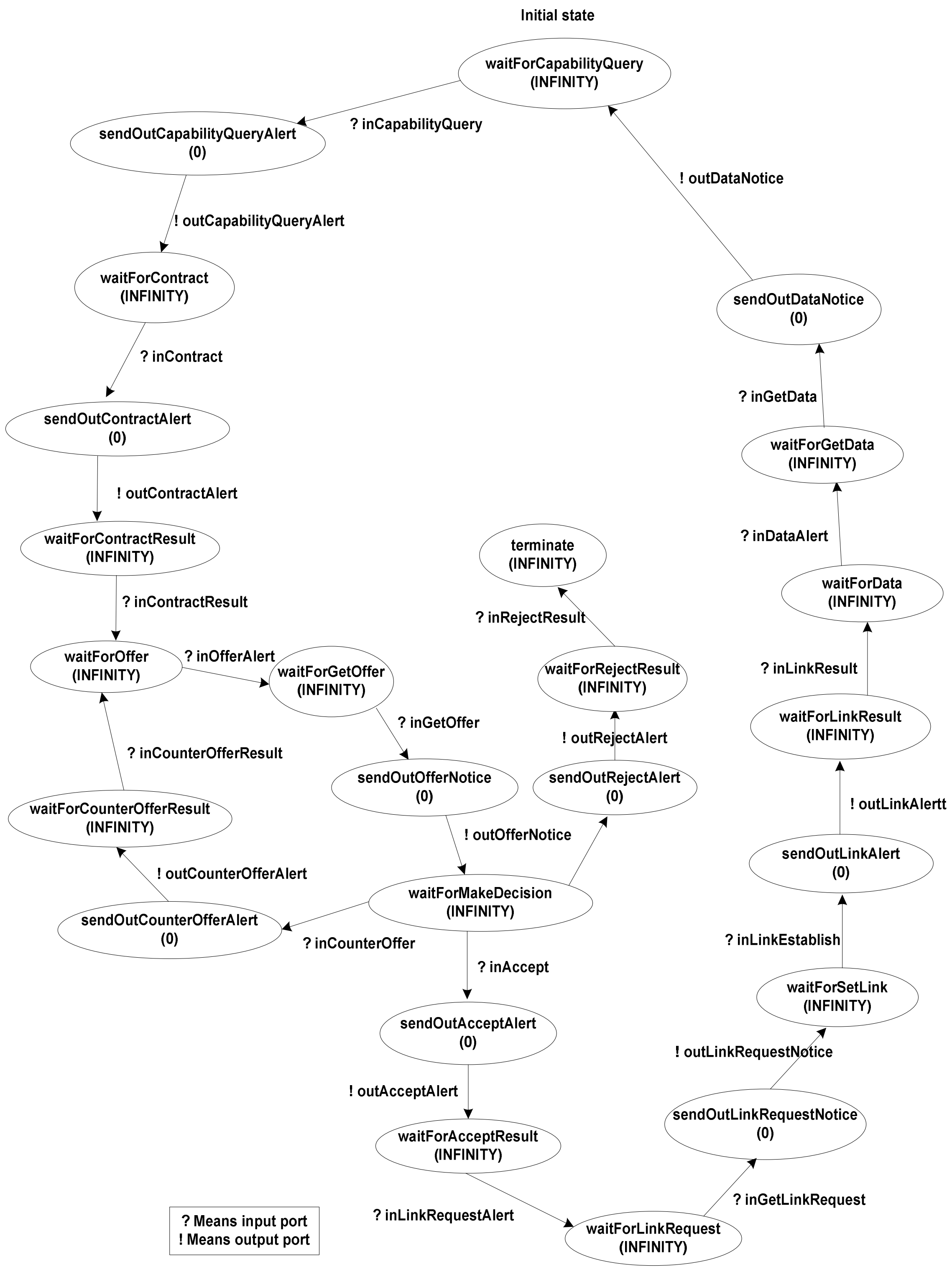


Figure 5-16 The state diagram for a user observer

Receiving the OfferAlert message means that the provider model sends the offer to the marketplace service, and the user model can pick it up from the marketplace service. To pick up the message, the user model invokes the operation called “GetOffer” and the information of invocation is sent to the DEVS agent model coupled with the user model. The DEVS agent sends the information to the user observer service. As soon as the user observer model in the user observer service receives the message from inGetOffer port, the user observer changes its state to “sendOutOfferNotice” where it produces an OfferNotice message to notify that the user model picked up the offer message to the provider observer. The state diagrams for the user observer and the provider observer are based on the behaviors of the user and the provider models as well as add ports and states for alerting messages, resulting messages, and noticing messages to communicate between the user observer and the provider observer.

Figure 5-17 represents the state diagram for the provider observer whose initial state is “waitForCapabilityQueryAlert” with infinity units. The provider observer has passive states until it receives a ContractAlert message. After that, it changes its state to “sendOutContractNotice” where it produces a ContractNotice message to let the user observer know that it received the ContractAlert message. The state diagram shows the states and ports having words like “Alert”, “Notice”, and “Result”.

The observer models watch behaviors of the user and the observer models which invoke the web service and exchange the information of their states. For instance, when the user model sends a message to the provider model through the marketplace web service, the user observer lets the provider observer know what the user does.

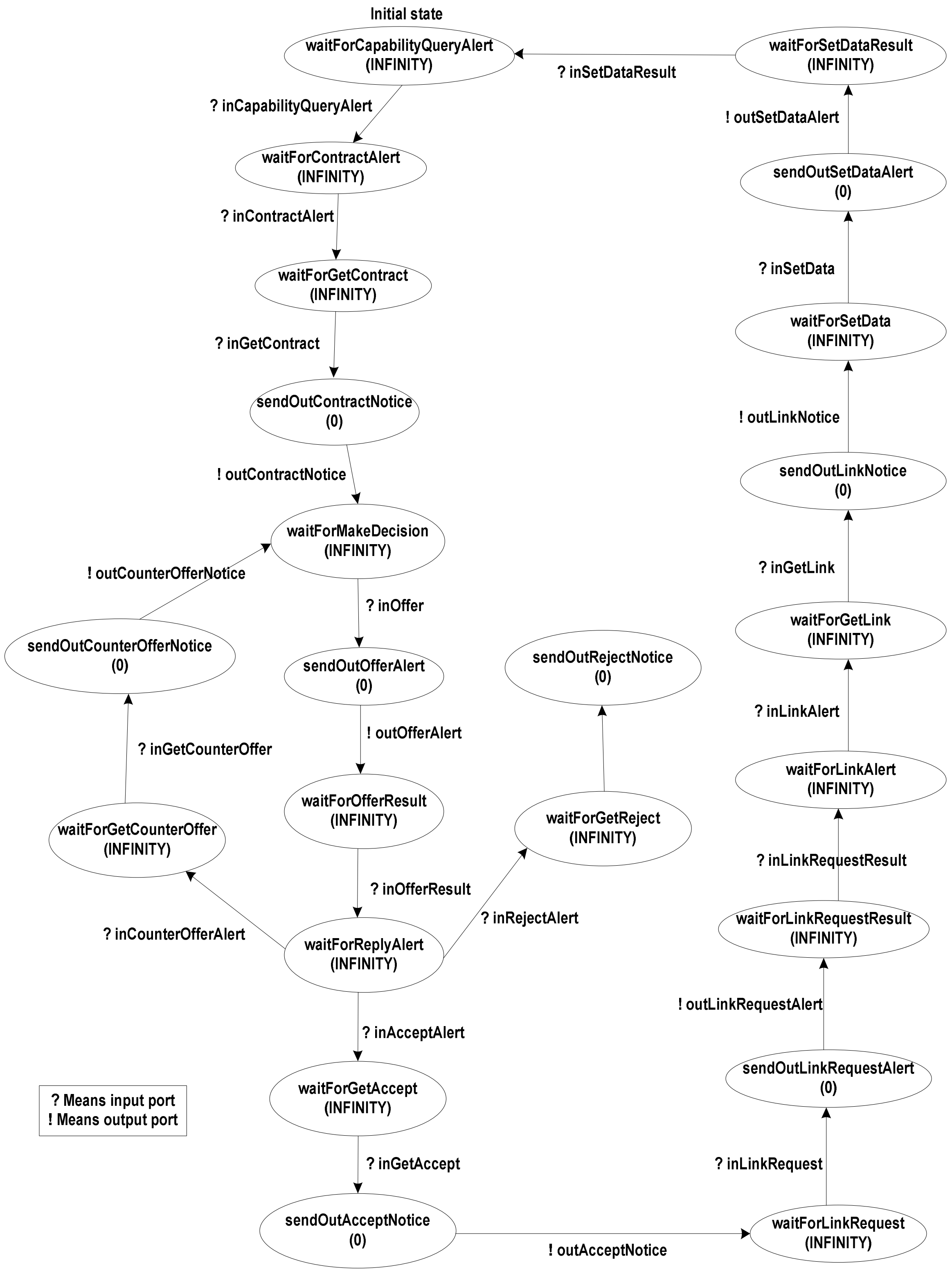


Figure 5-17 The state diagram for a provider observer

5.3.2. Implementation of Test Agents for Net-centric applying DEVS simulator service

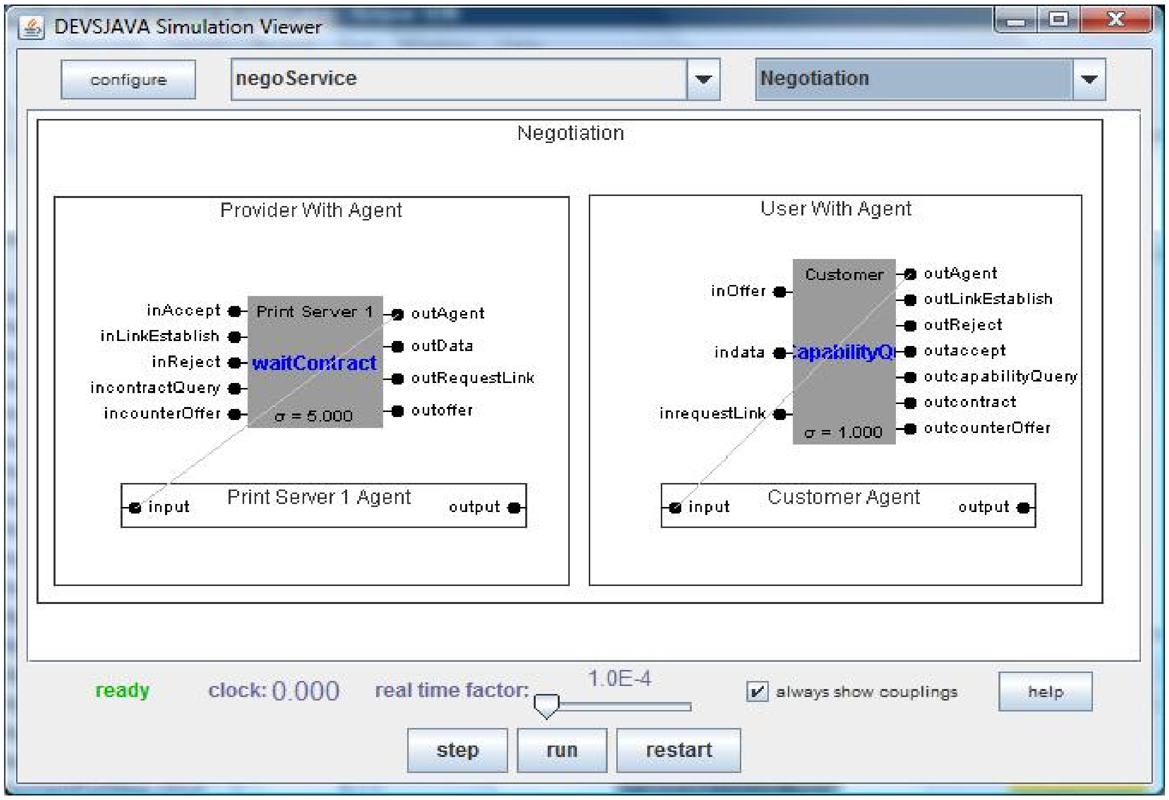


Figure 5-18 The DEVS model view of negotiation system

There are two implementations for testing agents for Net-centric. One is for implementing the user, the provider, and the marketplace service, and the other is for generating the user and the provider observer models.

Figure 5-18 represents the whole negotiation system with the DEVS agent. The user and the provider model are generated in accordance with the behaviors in the state diagrams. The DEVS agent consists of a DEVS coupled model, GUI for displaying results, as seen in figure 5-19. The DEVS coupled model has four atomic models such as the DevsServiceListener, agent Acceptor, Agent transducer, and observer agent. The DevsServiceListener has its state changed to active whenever a web service client invokes a web service. The role of an Agent transducer is to calculate statistical data for web service invocation and send it to an Agent Acceptor. The Agent Acceptor decides if the statistics from the agent transducer meets the predefined threshold values. The observer agent has a capability to send a message to the observer service using a receiveInput operation provided by the observer service. The receiveInput operation needs four arguments: a name of the observer model, an input port name, a message, and an output port name.

The DEVSAgent displays the results of the statistics for a web service invocation through a GUI composed of two tabs. One tab has a table for showing data and two text areas for showing request and response SOAP messages. The other is to write logs for invoking web services.

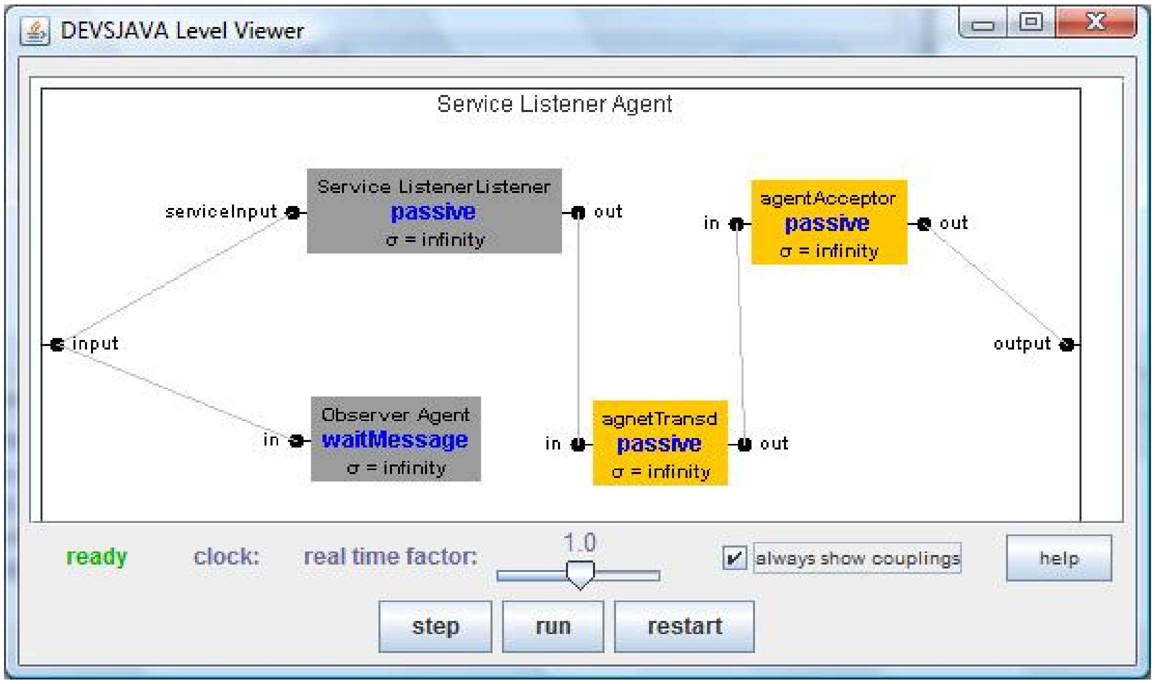


Figure 5-19 The four atomic models in DEVS Agent

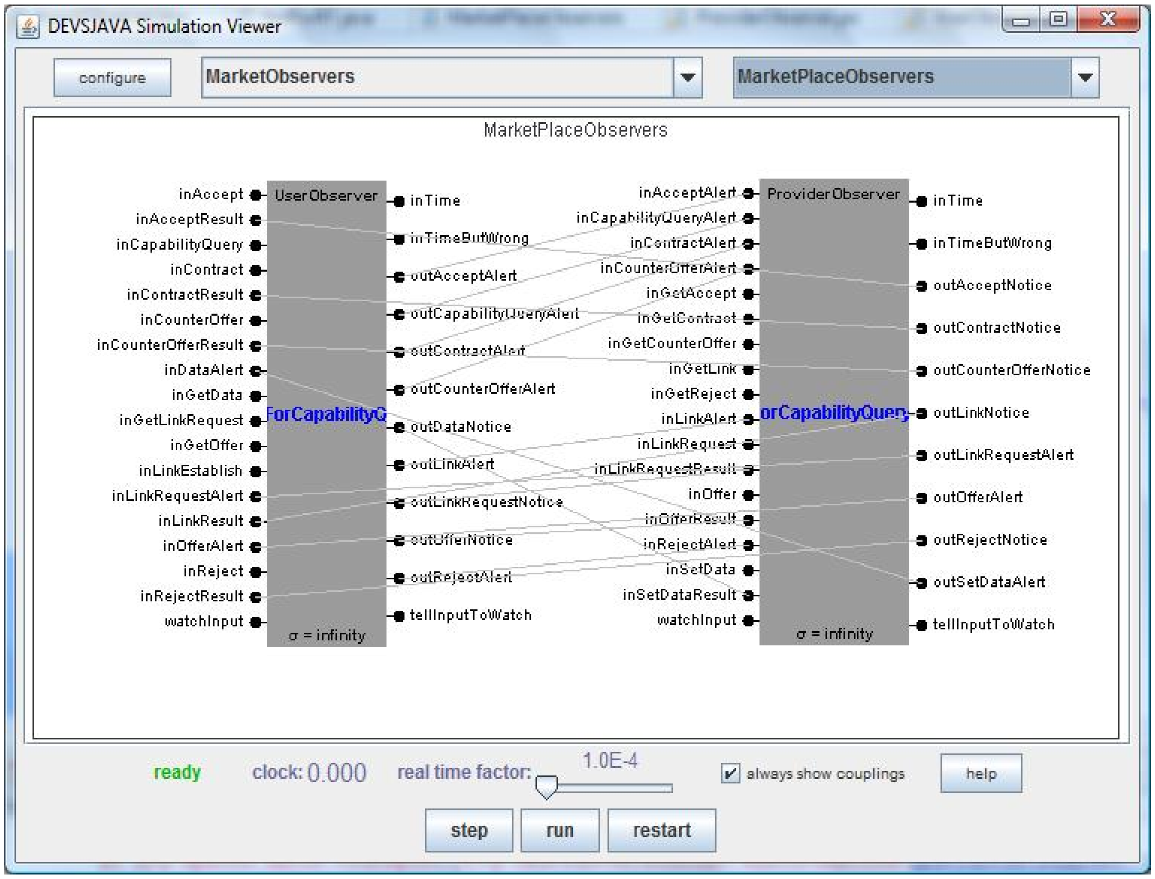


Figure 5-20 The view of observer models

The observer models, as shown in figure 5-20, consist of an UserObserver and a ProviderObserver model. The input/output ports have specific data types to integrate the UserObserver and the ProviderObserver using the DEVS simulator service integrator. The coupling between two observers is made through the ports whose names end with “Alert”, “Notice”, and “Result”. The others have the same data types shown in table 5-2.

Table 5-4 Messages to send/receive between the observer models

Name of Message	Name of Variable	Type of Variable	Output port Input port

CapabilityQueryAlert	customer, time	String	outCapabilityQueryAlert inCapabilityQueryAlert
ContractQueryAlert	customer, printJobID, time	String	outContractAlert inContractAlert
CounterOfferAlert	customer, printJobID, time	String	outCounterOfferAlert inCounterOfferAlert
OfferAlert	printServer, printJobID, time	String	outOfferAlert inOfferAlert
AcceptAlert	customer, printServer, printJobID, time	String	outAcceptAlert inAcceptAlert
LinkAlert	customer, content , time	String	outLinkAlert inLinkAlert
LinkRequestAlert	customer, printServer, time	String	outLinkRequestAlert inLinkRequestAlert
RejectAlert	customer, printServer, printJobID, time	String	outRejectAlert inRejectAlert
SetDataAlert	printServer, customer, time	String	outSetDataAlert inSetDataAlert
OfferNotice OfferResult	customer, printJobID, time	String	outOfferNotice inOfferResult
LinkRequestNotice LinkRequestResult	customer, printServer, time	String	outLinkRequestNotice inLinkRequestResult
DataNotice DataResult	printServer, customer, time	String	outSetDataNotice inSetDataResult
ContractNotice ContractResult	printServer, printJobID, time	String	outContractNotice inContractResult
CounterOfferNotice CounterOfferResult	printServer, printJobID, time	String	outCounterOfferNotice inCounterOfferResult
RejectNotice RejectResult	customer, printServer, printJobID, time	String	outRejectNotice inRejectResult
AcceptNotice AcceptResult	customer, printServer, printJobID, time	String	outAcceptNotice inAcceptResult
LinkNotice LinkResult	printServer, content , time	String	outLinkNotice inLinkResult

All messages have time variables to record the time to receive any message from the observers or the user and the provider model. With recorded time data, we can diagnose network delay and healthiness.

The observer models can be DEVS simulator services using DEVSJAVA and AXIS2, and are integrated by the DEVS simulator services integrator. The testing agents system requires the real time system. To support the real time simulation between observer services, The DEVS simulator service should equip the real time codes called *RTSimulator*. The *RTSimulator* simulates its model without interacting with the other RTSimulators for virtual time simulation protocol. Only the *RTSimulator* sends and receives a message from/to other *RTSimulator* in the coupling information.

Figure 5-21 depicts the overall testing agent system consisting of the marketplace service, DEVSJAVA model, and web enabled DEVS simulation environment which has DEVS simulator service and real time simulation. The marketplace and DEVSJAVA model have an invocation relation, that is, the model invokes an operation of marketplace. The characters on the blue arrows are operations in the marketplace service and the number is the order of the invocation. The DEVSJAVA model and the web enabled DEVS simulation have a receiveInput relation, that is, The DEVSJAVA model invokes a receiveInput operation in the observer service. The numbers in front of the input ports indicate which input port has a message from the DEVSJAVA after which the operation is invoked. For instance, if the getOffer is invoked in the user model, the user observer receives a message in the ingetOffer port. The ports connected between the observers are displayed in the center of the web enabled DEVS simulation.

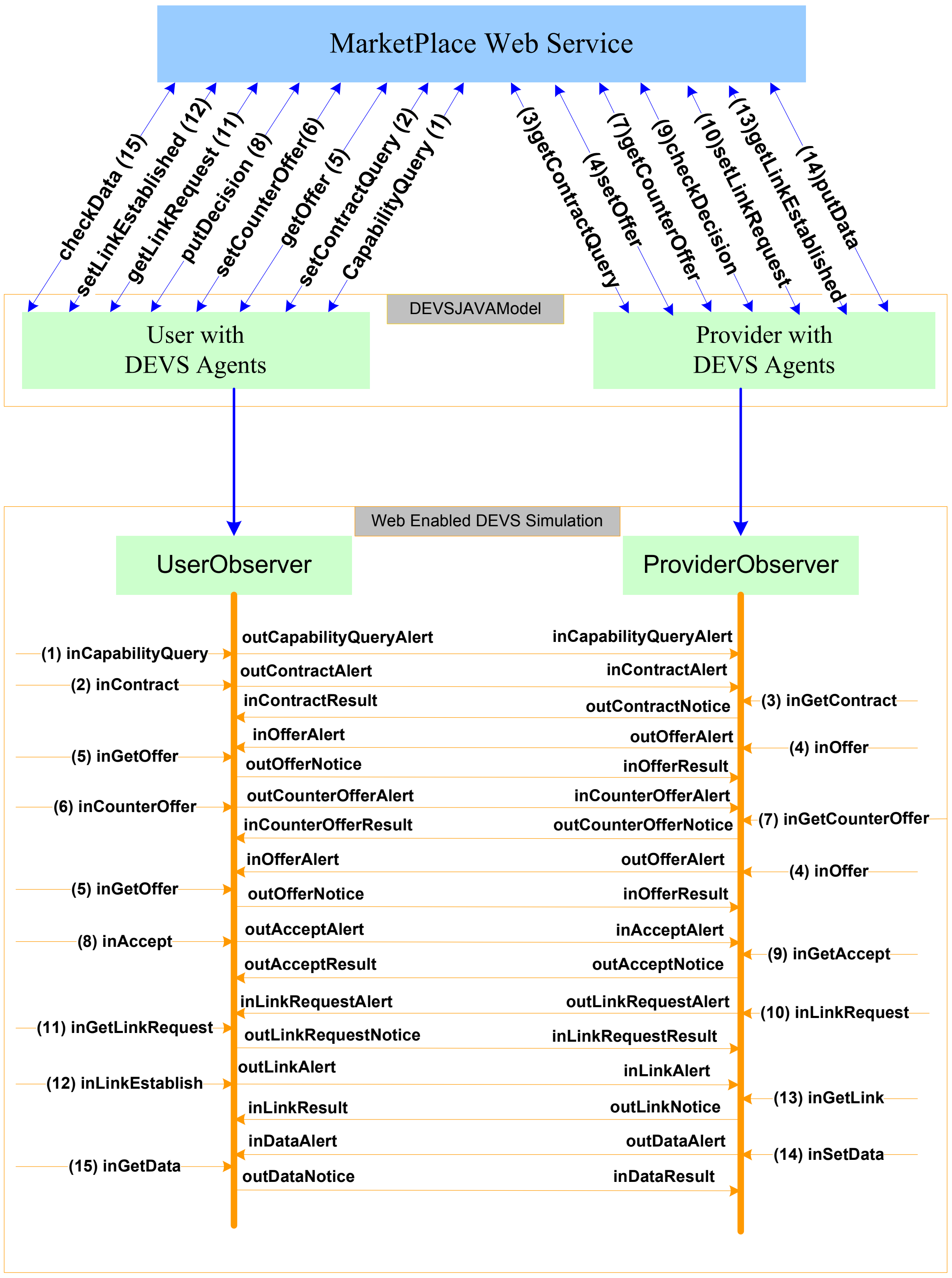
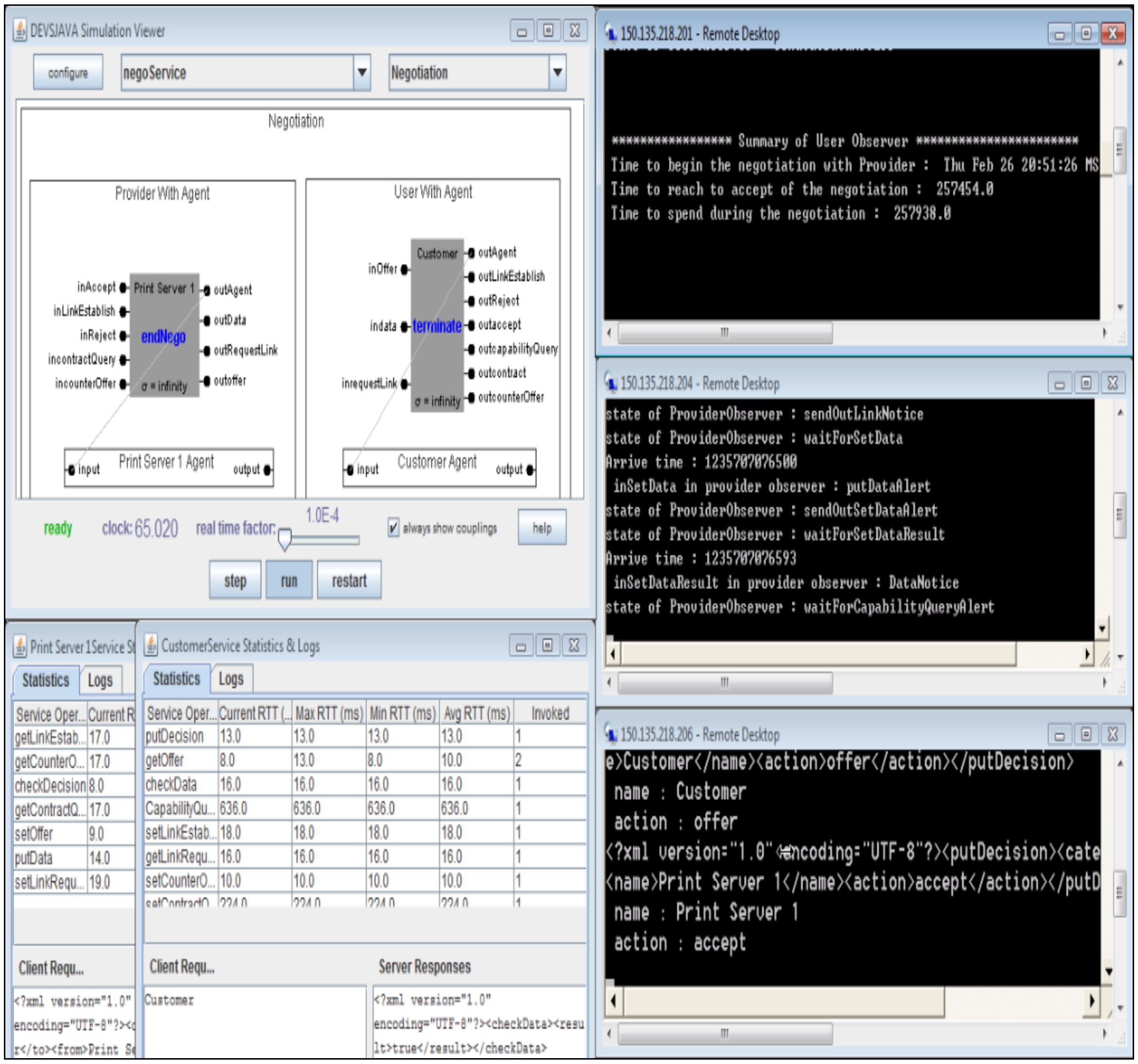


Figure 5-21 Overall testing agent system



```
***************** Summary of User Observer ***********************
Time to begin the negotiation with Provider :  Thu Feb 26 20:51:26 MS
Time to reach to accept of the negotiation :  257454.0
Time to spend during the negotiation :  257938.0
```

```
state of ProviderObserver : sendOutLinkNotice
state of ProviderObserver : waitForSetData
Arrive time : 1235707076500
 inSetData in provider observer : putDataAlert
state of ProviderObserver : sendOutSetDataAlert
state of ProviderObserver : waitForSetDataResult
Arrive time : 1235707076593
 inSetDataResult in provider observer : DataNotice
state of ProviderObserver : waitForCapabilityQueryAlert
```

```
e>Customer</name><action>offer</action></putDecision>
 name : Customer
 action : offer
<?xml version="1.0" encoding="UTF-8"?><putDecision><cate
<name>Print Server 1</name><action>accept</action></putD
 name : Print Server 1
 action : accept
```

Service Oper...	Current R
getLinkEstab...	17.0
getCounterO...	17.0
checkDecision	8.0
getContractQ...	17.0
setOffer	9.0
putData	14.0
setLinkRequ...	19.0

```
<?xml version="1.0"
encoding="UTF-8"?><c
r</to><from>Print Se
```

Service Oper...	Current RTT (...	Max RTT (ms)	Min RTT (ms)	Avg RTT (ms)	Invoked
putDecision	13.0	13.0	13.0	13.0	1
getOffer	8.0	13.0	8.0	10.0	2
checkData	16.0	16.0	16.0	16.0	1
CapabilityQu...	636.0	636.0	636.0	636.0	1
setLinkEstab...	18.0	18.0	18.0	18.0	1
getLinkRequ...	16.0	16.0	16.0	16.0	1
setCounterO...	10.0	10.0	10.0	10.0	1
setContractQ	224.0	224.0	224.0	224.0	1

```
Customer
```

```
<?xml version="1.0"
encoding="UTF-8"?><checkData><resu
lt>true</result></checkData>
```

Figure 5-22 The experiment of the testing agents system

The procedures for the experiment of the testing agents system are the following:

1. Integrate observer services with DEVS simulator services integrator.

2. Execute integrated observer services with *RTSimulators*.

3. Simulate the negotiation system with DEVS Agent.

Figure 5-22 shows the *simView* displaying the negotiation model, two windows displaying statistics and logs for invocation of the marketplace service, and three command windows displaying APACHE web servers. Table 5-5 represents the servers containing services and a client. The procedure 1 is executed in the 150.135.218.199 with the DEVS simulator services integrator. The procedure 2 is done by executing the XML document for the observer models. The *UserObserver* service is located in the 150.135.218.201, and the *ProviderObserver* service is located in the 150.135.218.204. After finishing the procedure 2, the *simView* is run to simulate the negotiation system with the DEVS agent. With the "step" button in the *simView*, we can simulate the negotiation system step by step. With the "run" button the simulation of the negotiation system is executed until the simulation is over.

Table 5-5 Servers, services, and negotiation model

Server name	Service and client
150.135.218.199	A user and provider with DEVS Agent, DEVS simulator services integrator
150.135.218.201	UserObserver service
150.135.218.204	ProviderObserver service
150.135.218.206	Marketplace service

Figure 5-22 displays the picture after the simulation is over. Each server displays the texts from its model or marketplace service. The observer servers display the texts for input message information and arrival time. The server with the UserObserver presents the summary of observation of the user model, and the servers with observer can report the negotiation outcome and duration. The windows for statistics and logs display current, average, max, and min round trip time (RTT) in the tables along with the number of invocation.

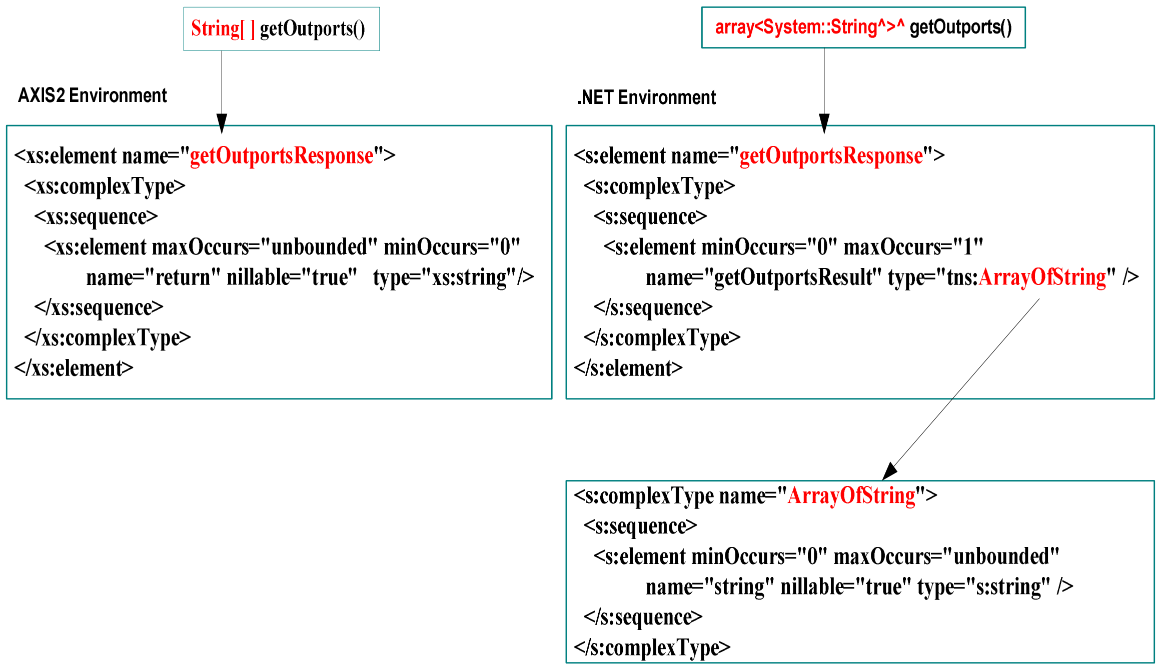
In testing the agents system, we see that the DEVS simulator services with observer models can be integrated with the particular group through the interoperability of message types. Different domain observer models can be integrated if they have an agreement on their messages sent/received to/from output ports/input ports. The message types are described in the DEVS namespace to verify that the coupled models are using the same syntactic structures.

We demonstrate the design concept of the testing agents system to prove that the testing agents system can test multiple levels with the negotiation system and its observer services. As a result of the experiment, the testing agents system can not only observe their observee models, such as a user and a provider model, but also it can assess the negotiation system from the network level to the pragmatic level with the DEVS Agent and observer services if the observer models have more diagnostic functionalities.

CHAPTER 6. DISCUSSIONS

6.1. Different WSDL with the same Design

When we design a web service, we focus on the definition of the operations and data types used as arguments and return values in the operations. For example, when we want to implement an operation that has a string argument and a string return in the web service, we can define the operation such as string getName(string). Obviously, the data types used in the operation are converted to schema in the WSDL by a tool provided in the web service middleware such as AXIS2 and .NET. Two middleware produce the same signature for operations in the web service if the data types are all primitive type. But when a complex data type is used in the operations, the tool converting a class used for a web service to a WSDL document produces different kinds of schema.



```
<xs:element name="getOutportsResponse">
  <xs:complexType>
    <xs:sequence>
      <xs:element maxOccurs="unbounded" minOccurs="0"
            name="return" nillable="true"   type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

```
<s:element name="getOutportsResponse">
  <s:complexType>
    <s:sequence>
      <s:element minOccurs="0" maxOccurs="1"
            name="getOutportsResult" type="tns:ArrayOfString" />
    </s:sequence>
  </s:complexType>
</s:element>
```

```
<s:complexType name="ArrayOfString">
  <s:sequence>
    <s:element minOccurs="0" maxOccurs="unbounded"
          name="string" nillable="true" type="s:string" />
  </s:sequence>
</s:complexType>
```

Figure 6-1 The data type conversion to schema in AXIS2 and .NET

Figure 6-1 represents each return data type of getOutports operation generated in AXIS2 and .NET. The meaning of the operation is that the operation gives string array value for outports. The signature of the method in AXIS2 is different from that of the method in .NET. But the name of the element tag of a return type of getOutports is the same. The return type is mapped to “getOutportsResponse”, which describes its data type using a element tag. *String array* in the AXIS2 is expressed to an element tag that has five attributes called maxOccurs, minOccurs, name, nillable, and type. AXIS2 assigns the values to the attributes with their rules. The name of *String array* is assigned to “return”, type is “xs:string”, and maxOccurs is “unbounded”. The *array<String>* in the .NET has different properties. The .NET defines a complexType called “ArrayOfString” to express array<String> type used in VC++. The ArrayOfString element is the same as the getOutportResponse element, except for the name attribute. In the case of the ArrayOfString element, the name is “string”.

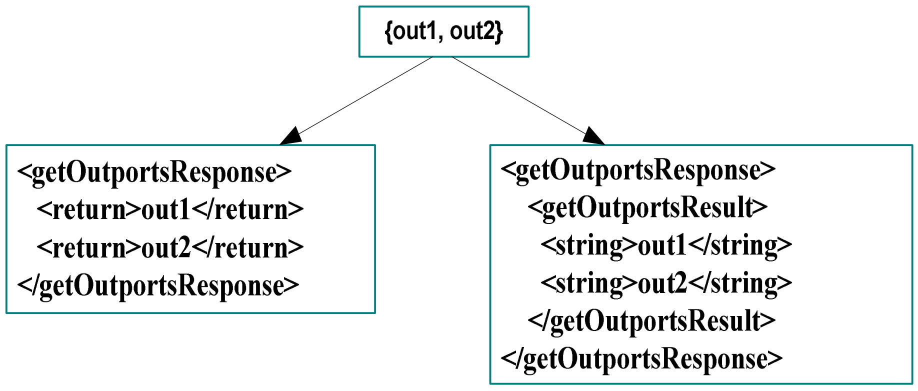


Figure 6-2 Instance of getOutportsResponse type

Figure 6-2 represents an instance of “getOutportsResponse” type in the AXIS or .NET.

The names of outports are “out1” and “out2”. AXIS2 generates SOAP body like the lift side of figure 6-2 when getOutports is invoked by a user, whereas .NET produces SOAP body, as shown on the right side of figure 6-2. There is a difference of expression of data types between AXIS2 and .NET.

Another difference is to define the namespace used in the WSDL. Target namespace can be defined by a user in the AXIS2 and .NET. AXIS2 provides flexible naming on namespace or schema namespace. The target namespace is important to invoke web services because an SOAP message contains the information of target namespace. If a different target namespace is used in the SOAP message to invoke the web service, the web server gives an error message complaining that the target namespace is not matched. AXIS2 allows a user to define target namespace for schema when we use data types assigned to classes.

We designed the DEVS simulator service with the expectation it be executed by a generic approach simulation program because we thought conceptually that the DEVS simulator service would provide homogeneous values although it is implemented in different languages and platforms. However, at the implementation level, different rules are applied to each development environment, such as AXIS2 and .NET. The generic simulation program is implemented in the DEVS simulator service integrator except for client codes containing *stub* classes to communicate with web services. If different web service middleware, except for AXIS2 and .NET, provides DEVS simulator service with a different definition of data types, the generic simulation program for DEVS simulator service should be modified to recognize client codes for new service. To distinguish where the DEVS simulator service is from, the simulation program uses the name of WSDL. The WSDL created by AXIS2 ends with “servicename?wsdl”, whereas the WSDL by .NET ends with “servicename.asmx?WSDL”. When the name of WSDL includes “asmx”, the service is invoked by client codes for .NET. If the name of WSDL does not include “asmx”, then client codes for AXIS2 are used to simulate DEVS simulator services.

6.2. XML message converter

XML to specific language instance conversion is introduced in chapter 5. In case of AXIS2 supporting a Java language, Java provides information of attributes from the class, a mechanism to make an instance of a class with a name of the class and invocation of a method with the method name and argument values. With these features, Java class defined by a user provides its information to other user or programs. However, there is one condition to be satisfied, that is, that an argument type be known. For example, assume that we define *void setName(String),* to invoke the *setName* method, we should know the type of argument. In the XML message, the information of the Java class is contained. So *XMLObjectMessageHandler* class can automatically convert XML message to Java class and vice versa.

XMLObjectMessageHandler can not cover all Java class because of lack of information on the class. The DEVS message used in the DEVS simulator service has a format to help *XMLObjectMessagHandler* convert the message. Figure 6-3 shows the example of a DEVS message with the format. Job class inherits entity class which is base class in the DEVS modeling. There are two variables and four methods to set/get variables. The set/get method is required to make an instance of a class from XML. There is a rule to make the set/get method, that is, “set/get” + “variable name” with first character written in capitals. For example, in case of id variable, the methods are “getId” and “setId”, as seen in figure 6-3. If other methods not following the above rules are added in the Job class, Job instance misses the information in the process of converting XML to Java instance. *XMLObjectMessageHandler* does not cover highly complex classes not following the rules.

```
Class Job extends entity{
        int id;
        double time;
        Job(){
             super(“Job”);
        }
        public int getId(){
             return id;
        }
        public void setId(int i){
             id =i;
        }
          public double getTime(){
             return time;
        }
        public void setTime(double t){
             time = t;
        }
}
```

Figure 6-3 The example DEVS message with the format

In case of .NET supporting VC++, the conversion XML to C++ instance is manually executed because C++ class does not have any base class which handles information of that class. There is no way to get the information of C++ class, such as the names and types of variables. Also, the mechanism to make an instance with a class name is required in the C++ to automatically convert C++ class to XML and vice versa.

An XML message from the DEVS simulator service for ADEVS should be parsed in the DEVS simulator service for DEVSJAVA. In our previous work, an XML message for DEVSJAVA is created using a XML handler called AXIOM (AXIs Object Model), whereas XML message for ADEVS is created using a DOM for C++. When running the integrated services, there is an error regarding XML parsing. The service for ADEVS does not recognize a XML document from the service for DEVSJAVA. Finally, the XML message for DEVSJAVA was generated using a DOM for Java, and the problem was solved.

6.3. Other issues

Other issues concern web services and web server platform. The issue for web service is that web service is stateless, which means the value of the variable in the web service does not continue in the next invocation. The DEVS simulator service needs to keep the variables for simulation. In current implementation, we use static variables for the server system to hold the variables in AXIS2 and .NET. It may cause errors when the same services are participating in the many integrated services.

When simulating the DEVS simulator services in the windows XP, there is an error in the connection refusal. This problem comes from a server platform using XP OS because XP OS has a long timeout period for socket connections. To solve this problem, the timeout period should be set to a shorter period in the registry in XP.

CHAPTER 7. CONCLUSIONS AND FUTURE WORK

7.1. Conclusions

As the request for reusability in the software industry increases, it is inevitable that the interoperability problem will occur. Interoperability requires platform independence and neutral message passing. SOA provides an interoperability environment satisfying the above requirements. DEVS modeling and simulation provides adaptability because DEVS theory can be implemented in any environment and system with various computer languages. Integration of DEVS and SOA gives interoperability environment to every domain.

In this study, we implemented an interoperable DEVS simulation environment using SOA and DEVS M&S. In the environment, SOA provides network interoperability and DEVS M&S provides message and pragmatic interoperability. DEVS simulator interoperability is implemented by DEVS simulator service consisting of three layers, that is, simulation protocol layer, message connection layer, and reporting layer. The simulation protocol layer provides basic functionality to simulate DEVS models. The message connection layer provides message type information to a DEVS simulator service integrator. Through this layer, heterogeneous DEVS simulator services can be integrated. The report layer provides the result of simulation of information generated during the simulation period.

SOA uses an SOAP message to provide an interoperable environment. When SOA and DEVS meet, the SOAP message gives fixed messages to DEVS models residing in the DEVS simulator services. To overcome this problem, we employ XML-style message passing on an SOA environment. An XML-style message passing means that the DEVS message in specific language is converted to a XML DEVS message. This XML DEVS message conforms to the message part of DEVS theory which defines a message as a set of pairs containing a port and a value. The value can be any type of class defined in the modeling. DEVS messages are converted to XML-style messages to be interoperable in the different language and platform. The Dynamic converter of JAVA object to XML message is implemented in the environment. But the converter does not cover all possible JAVA objects. In case of ADEVS, the XML message generation codes are inserted into the DEVS simulator service whenever the DEVS model is changed because C++ does not have a mechanism to dynamically get information of classes.

The ADEVS library does not cover some operations in DEVS simulator service because it does not provide those functions. We modified an ADEVS simulator in order to map the methods of the ADEVS simulator to operations of the DEVS simulator service. Through the extended ADEVS simulator, ADEVS models can be simulated with DEVSJAVA models.

To integrate heterogeneous DEVS simulator services, we developed a DEVS simulator service integrator which extracts information from WSDLs of the DEVS simulator services and verifies if two simulator web services have common messages during their coupling. As a result of integration of the services, a XML document is created and is utilized to execute integrated services. We demonstrate interoperable DEVS simulation using a GPT model implemented in AXIS2 and .NET. The *GPT* model consists of an *EF* model implemented in DEVSJAVA and a *Processe*r model implemented in ADEVS. Each model is embedded in its DEVS simulator service.

We designed and implemented the DEVS namespace which is schema document containing data types of DEVS messages. The DEVS namespace can be updated by a web service called “NamespaceService”. Through the *NamespaceService*, a service provider can register schema for message types used in a DEVS model in the DEVS namespace and look up schema to interoperate with a necessary model when the provider generates DEVS simulator services. As a result, each web service shares common message types. When integrating DEVS simulator services, the DEVS namespace provides a semantic interoperability between the DEVS simulator services.

We showed various applications of the interoperable DEVS simulation environment. The applications were drawn from real world development of automated testing environments for military information system interoperability. A radar track generation and display federation and a model negotiation web service illustrated the ability of the proposed middleware to work across platforms and languages. Its ability to support higher level semantic interoperability was demonstrated in a testing service that can deploy model agents to provide coordinated observation of web requests of participants in simulated distributed scenarios.

7.2. Future Work

In the future, we need to design and implement pragmatic level interoperability in the DEVS simulator service. The pragmatic level interoperability requires intentions of usage of the messages in the models. The DEVS simulator service should have the information for the pragmatic level interoperability. When it is integrated and executed with other services, each service checks its incoming messages for its pragmatic information interoperability. We need to define what pragmatic information is to apply to the interoperable DEVS simulation system. If the DEVS simulator service has pragmatic functions, it is possible to perform multi- levels testing suggested in [35].

We used two implementations, DEVSJAVA and ADEVS, of DEVS modeling and simulation to show demonstrations of the interoperable DEVS simulation system. To interoperate with other implementations of DEVS, the DEVS simulator service needs to be generated for them. For example, PythonDEVS, DEVSSim++, CD++, and DEVS Matlab are interoperable if the DEVS simulator service exists for them.

In agents system, we introduced a real time simulator in the DEVSJAVA. It needs to be added in the DEVS simulator service with the ADEVS. The DEVS simulator service with real time simulator can have autonomous functions to automatically search its corresponding services if information of services is given. It may require machine to machine communication and P2P concept to implement autonomous coupling and simulation.

The dynamic message conversion mechanism will be developed to make it easy to build up DEVS simulator services. Each language has different functions to extract information of its object. A JAVA language provides functions to get the names of variables and methods in the object and to invoke the methods. However, A C++ language does not provide those functions. In this case, we can add some functions which manipulate information of objects to the DEVS message class. The dynamic message conversion will be possible with the information of objects.

A web service for the DEVS namespace will be extended to delete data schema in the DEVS namespace. If there are some modifications on the DEVS message in the DEVS simulator service, the schema for the DEVS message should be updated to reflect new message types. In this case, the old schema is deleted and the modified schema is registered in the DEVS namespace.

REFERENCES

[1] Sage, A., “From Engineering a System to Engineering an Integrated System Family, From Systems Engineering to System of Systems Engineering”, 2007 IEEE International Conference on System of Systems Engineering (SoSE). April 16th -18th, 2007, San Antonio, Texas

[2] Jacobs, R.W. “Model-Driven Development of Command and Control Capabilities For Joint and Coalition Warfare,” Command and Control Research and Technology Symposium, June 2004.

[3] Muguira, J., Tolk., A “Applying a Methodology to identify Structural Variances in Interoperations,” JDMS: The Journal of Defense Modeling and Simulation, Vol 3, No 2, 2006

[4] Tolk, A., and Muguira, J.A. “The Levels of Conceptual Interoperability Model (LCIM)”, Proceedings Fall Simulation Interoperability Workshop, 2003

[5] DiMario M.J., “System of Systems Interoperability Types and Characteristics in Joint Command and Control”, Proceedings of the 2006 IEEE/SMC International Conference on System of Systems Engineering, Los Angeles, CA, USA - April 2006

[6] Levels of Information Systems Interoperability (LISI), http://www.sei.cmu.edu/isis/guide/introduction/lisi.htm

[7] Turnitsa C., and A. Tolk, “Evaluation of the C2IEDM as an Interoperability-Enabling Ontology,” Proceedings of Fall Simulation Interoperability Workshop, 2005.

[8] Zeigler, B.P., Fulton, D., Hammonds, P., Nutaro, J., “Framework for M&S Based System Development and Testing in Net-centric Environment”, ITEA Journal, Vol. 26, No. 3, October 2005

[9] Wutzler, T. H.S. Sarjoughian (2007), “Interoperability among Parallel DEVS Simulators and Models Implemented in Multiple Programming Languages”, SIMULATION: Transactions of The Society for Modeling and Simulation International, Accepted.

[10] Sarjoughian, H. S., and B. P. Zeigler. "DEVS and HLA: Complementary Paradigms for Modeling and Simulation?" Simulation: Transactions of the Society for Modeling and Simulation International 17, no. 4 (2000): 187-97.

[11] Mittal, S., and J. L. R. Martín. "DEVSML: Automating DEVS Execution over SOA Towards Transparent Simulators Special Session on DEVS Collaborative Execution and

Systems Modeling over SOA." Paper presented at the DEVS Integrative M&S Symposium DEVS' 07 2007.

[12] SOA http://www.sun.com/products/soa/index.jsp

[13] Web Service Architecture http://www.w3.org/TR/ws-arch/

[14] WSDL2.0 http://www.w3.org/TR/wsdl20-primer/

[15] SOAP1.2 http://www.w3.org/TR/soap12-part0/

[14] Zeigler, B.P., Kim, T.G., and Praehofer, H., Theory of Modeling and Simulation, 2nd ed., Academic Press, New York, 2000.

[15]. B. P. Zeigler, H.S. Sarjoughian, "Approach and Techniques for Building Component-based Simulation ModelsThe Interservice/Industry Training", presentation at Simulation and Education Conference '04, Orlando, FL

[16] Eric Newcomer and Greg Lomow, "Understanding SOA with Web Services", Addison-Wesley Professional, 2004

[17] D Box, D Ehnebuske, G Kakivaya, A Layman, "Simple Object Access Protocl (SOAP) 1.1", 2003

[18] James Snell, Doug Tidwell, and Pavel Kulchenko, "Programming Web Services with SOAP", O'Reilly Media, Inc.; 1 edition, 2001

[19] Thomas Erl, "Service-Oriented Architecture (SOA): Concepts, Technology, and Design", Prentice Hall PTR, 2005

[20] Apache AXIS2 : http://ws.apache.org/axis2/

[21] Turnitsa C., and Tolk, A., "Evaluation of the C2IEDM as an Interoperability-Enabling Ontology," Proceedings of Fall Simulation Interoperability Workshop, 2005.

[22] Lasschuyt , E., Henken, M., Treurniet, W., and Visser, M., "How to Make an Effective Information Exchange Data Model," RTO-IST-042/9,2004

[23] Hoffmann, M., "Challenges of Model Interoperation in Military Simulations". SIMULATION, Vol. 80, pp. 659-667, 2004

[24] Chaum, E., Hieb, M.R., and Tolk, A. "M&S and the Global Information Grid," Proceedings Interservice/Industry Training, Simulation and Education Conference (I/ITSEC), 2005.

[25] Zeigler, B.P. and P.E. Hammonds, *Modeling & Simulation-Based Data Engineering: Introducing Pragmatics into Ontologies for Net-Centric Information Exchange*. 2007.

[26] Zeigler, B.P., Mittal, S., Hu, X., “Towards a Formal Standard for Interoperability in M&S/Systems of Systems Engineering”, Critical Issues in C4I, AFCEA-George Mason University Symposium, May 2008

[27] DEVSJAVA : http://www.acims.arizona.edu/

[28] ADEVS: an open source C++ DEVS Simulation engine. Available at: http://www.ornl.gov/~1qn/adevs/index.html

[29] Microsoft Corporation. XML and .NET White Papers. http://www.microsoft.com/serviceproviders/whitepapers/xml.asp

[30] Xiaolin Hu, Bernard Zeigler, " A Proposed DEVS Standard: Model and Simulator Interfaces, Simulator Protocol"

[31] Mittal, S., Risco-Martín, J.L., Zeigler, B.P.,"Implementation of Formal Standard for Interoperability in M&S/Systems of Systems Integration with DEVS/SOA", submitted to C2 Journal

[32] Pullen, M., Wilson, L.T.C.K, Hieb, M., Tolk, A., “Extensible Modeling and Simulation Framework (XMSF) C4I Testbed,” available from http://www.movesinstitute.org/xmsf/xmsf.html

[33] Dahmann, J.S., F. Kuhl, and R. Weatherly, Standards for Simulation: As Simple As Possible But Not Simpler The High Level Architecture For Simulation. Simulation, 1998. 71(6): p. 378

[34] Mittal, S., Zeigler, B.P., Martin, J.L.R., Sahin, F., Jamshidi, M., “Modeling and Simulation for Systems of Systems Engineering”, to appear in Systems of Systems -- Innovations for the 21st Century (to be published by Wiley)

[35] Zeigler, B.P., and Hammonds, P., “Modeling & Simulation-Based Data Engineering: Introducing Pragmatics into Ontologies for Net-Centric Information Exchange”, 2007, New York, NY: Academic Press.

[36] Zeigler, B. P., Kim, T.G., and Praehofer, H., “Theory of Modeling and Simulation” New York, NY, Academic Press, 2000.

[37] Mittal, S., Risco-Martin, J.L., Zeigler, B.P. “DEVS-Based Web Services for Net-centric T&E”, Summer Computer Simulation Conference, 2007

[38] Badros, G. “JavaML: a Markup Language for Java Source Code”, Proceedings of the 9th International World Wide Web Conference on Computer Networks: the international journal of computer and telecommunication networking, pages 159-177

[39] Zeigler, B. P., Mittal, S., “Enhancing DoDAF with DEVS-Based System Life-cycle Process”, IEEE International Conference on Systems, Man and Cybernetics, Hawaii, October 2005

[40] Reichenthal, S.W., SRML - Simulation Reference Markup Language W3C Note 18 December 2002 http://www.w3.org/TR/SRML/

[41] Mittal, S., “Extending DoDAF to allow DEVS-Based Modeling and Simulation”, Special issue on DoDAF, Journal of Defense Modeling and Simulation (JDMS), Vol 3. No. 2

[42] Mittal, S. Martin, J.L.R., “Design and Analysis of Service Oriented Architectures using DEVS/SOA-Based Modeling and Simulation”, whitepaper at www.duniptechnologies.com

[43] Mittal, S., Martin, J.L.R., Zeigler, B.P., ”DEVS/SOA: A Cross-platform Framework for Net-centric Modeling and Simulation in DEVS Unified Process”, SIMULATION: Transactions of SCS, to appear

[44] Mittal, S., Martin, J.L.R., Zeigler, B.P., “*DEVSML: Automating DEVS Execution over SOA Towards Transparent Simulators*”, Special Session on DEVS Collaborative Execution and Systems Modeling over SOA, DEVS Integrative M&S Symposium DEVS' 07, Spring Simulation Multi-Conference, March 2007

[45] Mittal, S., Zeigler, B.P., Hwang, M.H., XML-Based Finite Deterministic DEVS (XFD-DEVS); http://www.saurabh-mittal.com/fddevs/

[46] ACIMS software site: http://www.acims.arizona.edu/SOFTWARE/software.shtml

[47] Hu, X., and Zeigler, B.P., “*Model Continuity in the Design of Dynamic Distributed Real-Time System”s*, IEEE Transactions on Systems, Man And Cybernetics— Part A, Volume 35, Issue 6, pp. 867-878, November 2005

[48] Cho, Y., Zeigler, B.P., Sarjoughian, H., “Design and Implementation of Distributed Real-Time DEVS/CORBA”, IEEE Sys. Man. Cyber. Conf., Tucson, Oct. 2001.

[49] Wainer, G., Giambiasi, N., “Timed Cell-DEVS: modeling and simulation of cell-spaces”. Invited paper for the book Discrete Event Modeling & Simulation: Enabling Future Technologies, Springer-Verlag 2001

[50] Zhang, M., Zeigler, B.P., Hammonds, P., “DEVS/RMI-An Auto-Adaptive and Reconfigurable Distributed Simulation Environment for Engineering Studies”, ITEA Journal, July 2005

[51] Mittal, S., “DEVS Unified Process for Integrated Development and Testing of Service Oriented Architectures”, Ph. D. Dissertation, University of Arizona

[52] DUNIP: A Prototype Demonstration http://www.acims.arizona.edu/dunip/dunip.avi

[53] MatLab Simulink, http://www.mathworks.com/products/simulink/

[54] OMNET++, http://www.omnetpp.org/

[55] NS-2, http://www.isi.edu/nsnam/ns/

[56] XDEVS web page: http://itis.cesfelipesegundo.com/~jlrisco/xdevs.html

[57] HLA, https://www.dmso.mil/public/transition/hla/

[58] Sarjoughian, H.S., Zeigler, B.P., "DEVS and HLA: Complimentary Paradigms for M&S?" Transactions of the SCS, (17), 4, pp. 187-197, 2000

[59] Carstairs, D.J., “Wanted: A New Test Approach for Military Net-Centric Operations”, Guest Editorial, ITEA Journal, Volume 26, Number 3, October 2005

[60] Mittal, S., Zeigler, B.P., “*DEVS* Unified Process for Integrated Development and Testing of System of Systems”, Critical Issues in C4I, AFCEA-George Mason University Symposium, May 2008

[61] Sarjoughian, H., Zeigler, B.P., and Hall, S., “A Layered Modeling and Simulation Architecture for Agent-Based System Development”, Proceedings of the IEEE 89 (2); 201-213, 2001

[62] HTTP : http://www.w3.org/Protocols/

[63] SMTP : http://cr.yp.to/smtp.html

[64] Mittal, S., Zeigler, B.P., Hammonds, P., Veena, M., “Network Simulation Environment for Evaluation and Benchmarking HLA/RTI Experiments”, JITC Report, Fort Huachuca, December 2004.

[65] Hu, X., Zeigler, B.P., Mittal, S., “Dynamic Configuration in DEVS Component-based Modeling and Simulation”, SIMULATION: Transactions of the Society of Modeling and Simulation International, November 2003

[66] Mittal, S., Zeigler, B.P.,, "Modeling/Simulation Architecture for Autonomous Computing", Autonomic Computing Workshop: The Next Era of Computing, Tucson, January 2003.

[67] XML: http://www.w3.org/XML/

[68] Martin, J.L.R., Mittal, S., et.al, "Optimization of Dynamic Data Types in Embedded Systems using DEVS/SOA-based Modeling and Simulation", 3rd International ICST Conference on Scalable Information Systems, Italy, June 2008

[69] aDEVS: an open source C++ DEVS Simulation engine. Available at: http://www.ornl.gov/~1qn/adevs/index.html

[70] Mittal, S., Martin,J.L.R., Zeigler, B.P., "WSDL-Based DEVS Agent for Net-Centric Systems Engineering", International Workshop on Modeling and Applied Simulation, Italy, September 2008

[71] Department of Defense Architecture Framework (DoDAF) version 1.5 downloadable from: http://www.defenselink.mil/cio-nii/docs/DoDAF_Volume_II.pdf

[72] Thea Clark, Richard Jones, "Organisational Interoperability Maturity Model for C2", 1999

[73] eclipse : http://www.eclipse.org/

[74] Moath Jarrah, " An Automated Methodology for Negotiation Behaviors in Multi-Agent Engineering Applications", summer 2008, ECE, University of Arizona

[75] Jean-Sébastien Bolduc and Hans Vangheluwe. The modelling and simulation package PythonDEVS for classical hierarchical DEVS. MSDL technical report MSDL-TR-2001-01, McGill University, June 2001.