Proceedings of the 2009 Winter Simulation Conference
M. D. Rossetti, R. R. Hill, B. Johansson, A. Dunkin and R. G. Ingalls, eds.

DEVS NAMESPACE FOR INTEROPERABLE DEVS/SOA

Chungman Seo
Bernard P. Zeigler

Arizona Center for Integrative Modeling and Simulation
The University of Arizona
1230 E. Speedway Blvd., Tucson, AZ 85721

ABSTRACT

Interoperable DEVS modeling and simulation is desirable to enhance model composability and reusability with DEVS models and non-DEVS models in different languages and platforms. The barrier to such interoperability is that DEVS simulators implement the DEVS modeling formalism in diverse programming environments. Although the DEVS formalism specifies the same abstract simulator algorithm, different simulators implement the same abstract simulator in different ways. This situation inhibits interoperating DEVS simulators and prevents simulation of heterogeneous models. Service Oriented Architecture provides a flexible approach to interoperability than because it provides platform independence and employs platform-neutral message passing with Simple Object Access Protocol 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 message namespace. The interoperable DEVS environment consists of a DEVS simulator service and an associated integrator. Using the DEVS namespace, DEVS simulator services can be interoperable with other such services using the same message types.

1 INTRODUCTION

The study of interoperability concerns methodologies to interoperate different systems distributed over a network system. Such a System of Systems (SoS) is differentiated from a single monolithic system in that it requires interoperability among its constituent systems (Sage 2007). Levels of interoperability describing technical interoperability and the complexity of interoperations have been suggested (Tolk et al. 2006, DiMario 2006) and are interpreted in different applications such as telecommunication and command and control software (Jacobs 2004). Zeigler and Hammonds (2007) introduced linguistic concepts of interoperability at three basic levels: pragmatic, semantic, and syntactic. The pragmatic level focuses on the receiver's interpretation of messages in the context of application relative to the sender's intent. The semantic level concerns definitions and attributes of terms and how they are combined to provide shared meaning to messages. The syntactic level focuses on a structure of messages and adherence to the rules governing that structure. The linguistic interoperability concept supports simultaneous testing environment at multiple levels (Zeigler et al. 2005).

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 (Sarjoughian and Zeigler 2000; Seo et al. 2004; Cheon et al. 2004; Mittal and Martin 2007; Wutzler and Sarjoughian 2007). These 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.

The Discrete Event System Specification (DEVS) is a formalism describing entities and behaviors of a system (Zeigler et al. 2000). A coupled model is made up of component models, and the coupling relations which establish the desired communication links. A coupled model specifies how to connect several component models together to form a new model. Two significant activities involved in coupled models are specifying its component models and defining the couplings which create the desired communication networks. The DEVS Protocol (Zeigler, Kim, and Praehofer 2000) specifies the abstract simulation engine that correctly simulates DEVS atomic and coupled models. Interpreted in a distributed simulation context, the DEVS abstract simulator gives rise to a general protocol that has specific mechanisms for declaring who takes part in the simulation (the federates). It also specifies how federates interact in an iterative cycle that controls how time advances, when federates exchange messages, and do internal state updating. A significant feature, in comparison to simulation based on the

HLA standard (Kuhl, Weatherly, and Dahmann 1999) is that if the federates in simulation are DEVS compliant then the simulation can be proved to be correct in the sense that the DEVS closure under coupling theorem guarantees a well-defined resulting structure and behavior (Zeigler, Kim, and Praehofer 2000).

DEVS modeling and simulation research groups have been interested in DEVS interoperability in order to enhance model composability and reusability with DEVS models and non-DEVS models in different languages and platforms (Kim and Kim 2005, Mittal and Martin 2007). 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 (Zeigler 2004), CD++ (Wainer 2002), smallDEVS(Janousek and Kironsky 2006)). Although 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. Nevertheless, different platforms specialize in different capabilities, e.g., C++ supports fast execution, but JAVA provides a better platform for web service development. Therefore, to exploit these diverse capabilities requires a capability to interoperate them at the abstract simulator level (see Section 5 for an example).

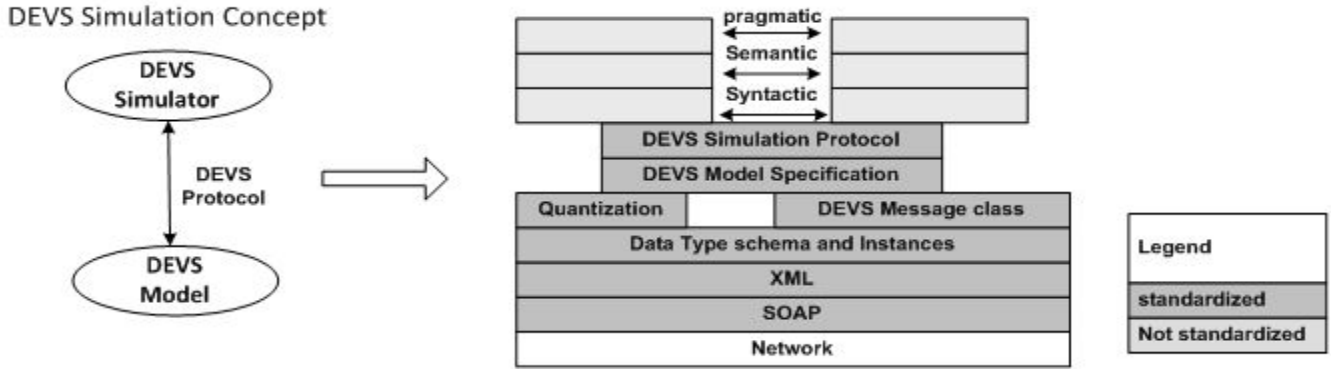

Figure 1: DEVS Standardization Supports Higher Level Web-Centric Interoperability

The demand for interoperable DEVS simulation has spurred the development of standards to interoperate DEVS models generated in the different languages and platforms (Vangheluwe et al. 2001). Figure 1 indicates that DEVS standardization can be sought at both the modeling level and at the simulator level to achieve higher level web-centric interoperability. The left side of the Figure 1 represents the well-known DEVS simulation concept in which a DEVS abstract simulator interacts with a DEVS model through the DEVS protocol. The right side of Figure 1 represents extended DEVS simulation concept for the distributed simulation on the network. The messages passed on the network are carried through SOAP messages. At the top, the three levels of interoperability mentioned above are envisioned (Zeigler and Hammonds 2007). Interoperability of DEVS models at the modeling level is the most challenging since it requires exposing the DEVS defining methods and data types of different languages to a common interface (Touraille, Traoré, and Hill 2009).

In contrast, DEVS interoperability at the simulator level requires exposing the defining methods of the DEVS abstract simulator so that different simulators can interface to the same DEVS protocol. Some research on DEVS interoperability at the simulator level has been studied along with HLA and SOA. Prior works include DEVS/SOA which Mittal and Rico developed using web services (Mittal and Martin 2007). Kim (2008) and Jarrah (2008) extended DEVS/SOA to run in real time but did not address interoperability. However, DEVS/SOA provides only platform interoperability because it employs JAVA serialization which converts JAVA objects into byte arrays to send messages to other simulators. This restricts interoperation to simulators based on JAVA. To add the language interoperability to the platform interoperability, we must enable platform neutral message passing and the SOA environment. The interoperability on DEVS at the simulator level interoperability uses common simulator interfaces to simulate DEVS models. The simulator interface describes a minimum agreement being able to implement a DEVS 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 by DEVS models in the DEVS simulator services. It provides semantic interoperability when we integrate different DEVS simulators.

In the rest of the paper, the background of SOA and web service is discussed in the section 2. The section 3 addresses an overall architecture of interoperable DEVS simulator services, DEVS namespace, design of the DEVS simulator service, and DEVS message to XML message. The section 4 explains implementation of the DEVS namespace and DEVS simulator services**.** The example of integration of web service is presented with DEVS simulator services embedding atomic DEVS mod-

els to create a track display coupled model in the section 5. A comparison between interoperability of DEVS/SOA with HLA-based DEVS interoperability is presented in section 6. The paper's summary and future work is in the section 7.

2 BACKGROUND: Service Oriented Architecture (SOA) and Web Service

SOA (Newcomer and Lomow 2004) 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.

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) (Box 2003). 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 platform independence.

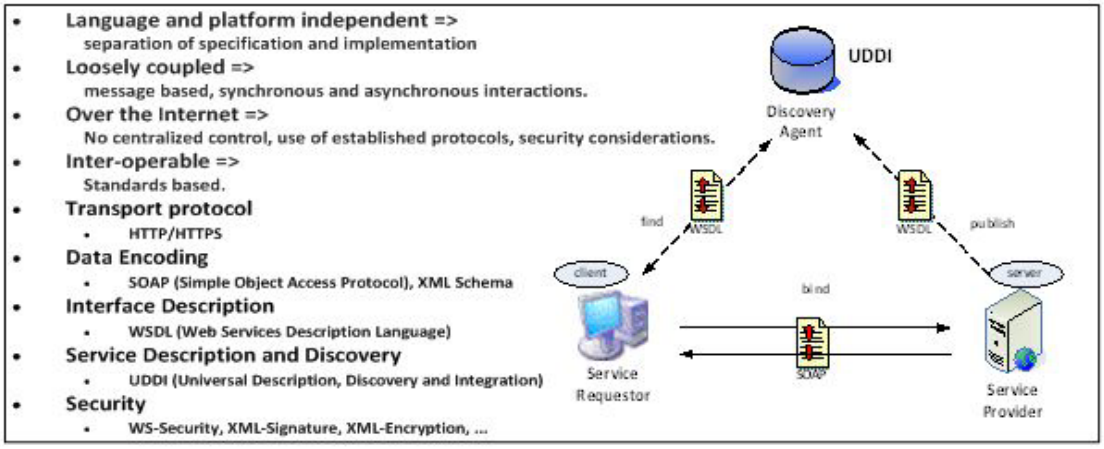

Figure 2: Web Services Architecture approach to build software application that use services available in a network

As seen in Figure 2, the web services technologies architecture 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) 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.

3 OVERALL ARCHITECTURE OF INTEROPERABLE DEVS SIMULATOR SERVICES

The system of interoperable DEVS simulator services is based on web technology and a DEVS namespace concept. The web service provides common infrastructure of system/language interoperability and the DEVS namespace presents a look-up table for messages that are passed between 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 (Zeigler, Mittal, and Hu 2008). The message types of each service are registered in the DEVS namespace before the service publishes in the server.

In Figure 3, two DEVS simulator services provide common interfaces on the different platforms. A common interface contains operations for the 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 (Seo and Zeigler 2009) 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 it easy to handle the XML. In the Figure 3, 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.

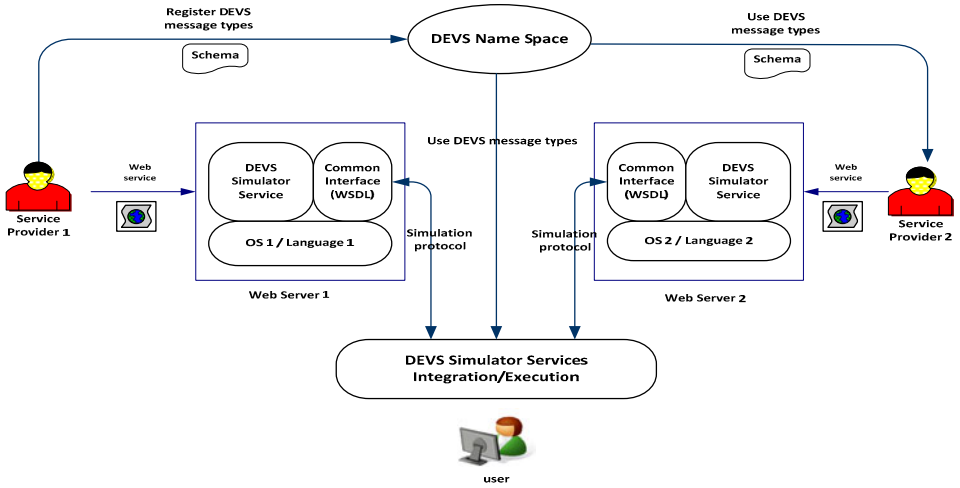

Figure 3: Overall system of interoperable DEVS simulator services

Figure 4 represents web-enabled interoperability of DEVS components which are made of different languages such as Java and C++. There are two services called aDEVS federate and DEVSJAVA federate, one DEVSJAVA client and DEVS namespace. The aDEVS federate is a DEVS simulator service encapsulating aDEVS modeling and simulation on the .Net environment (Seo and Zeigler 2009). However, the DEVSJAVA federate consists of DEVSJAVA and AXIS2 environment. The DEVSJAVA client is DSSIE which integrates DEVS simulator services and executes the integration. The DEVS namespace stores schemata for entity classes in messages. The federates can register and discover schemata for information exchange. SOAP messages are passed among the services and the client during the simulation.

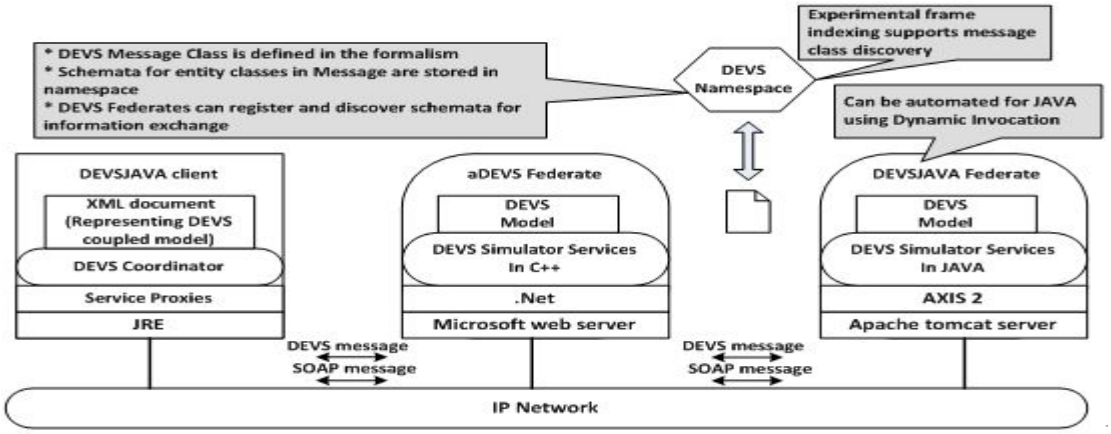

Figure 4: Web-enabled interoperability of DEVS components

3.1 DEVS Namespace

The WSDL for a DEVS simulator service defines data types used by each operation. When the web service communicates with a user, the operations of the web service receive an argument as an 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.

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 an indicator of a schema document 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 clarity for message passing between systems on interoperable operation.

Figure 5 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.

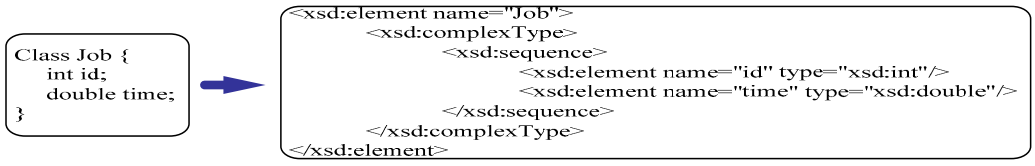

Figure 5: Conversion of *Job* class to schema data type

Conversion of a class to a schema is performed by a service provider. The schema document resulting from the conversion is registered into a DEVS namespace storage to access through the network. The procedure of registering a schema document 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 storage. Another operation, called *registerSchema*, is for registering the schema document to the DEVS namespace storage. The *getDomains* and *getMessageTypes* operations are used to search a schema document in the DEVS namespace storage and get a specific schema document.

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 execute the DEVS model as a component in a coupled model 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 be informed of 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.

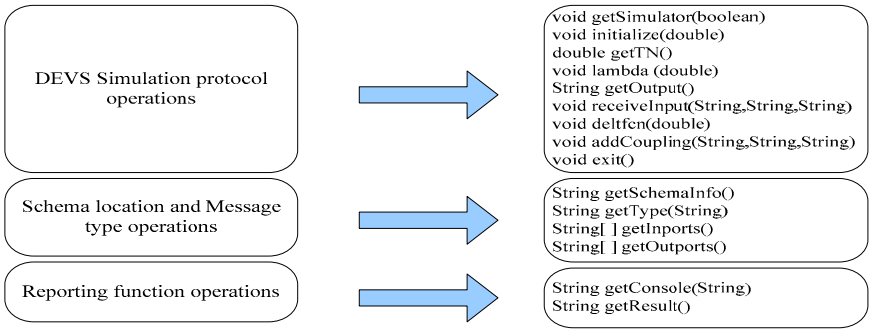

Figure 6: The operations of DEVS simulator service

As a result of all considerations, a DEVS simulator service has three categories: DEVS simulation protocol operations, schema location and message type operations, and reporting function operations. Figure 6 represents three categories of operations and signatures of operations.

The operations for DEVS simulation protocol at the top of the Figure 6 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 in Figure 3. 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 reaching the number of simulation protocol cycles.

The operations for schema location and message type in middle of the Figure 6 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 6 has two operations, that is, *getConsole* and *getResult*. The *getConsole* operation returns a document produced by the simulator service during a simulation protocol cycle. 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.

3.3 DEVS Message to XML Message

DEVS messages are defined as pairs consisting of a port and a value in the DEVS modeling and simulation. Implementations of the DEVS theory use these 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 7: The structure of the XML message

Figure 7 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 *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.

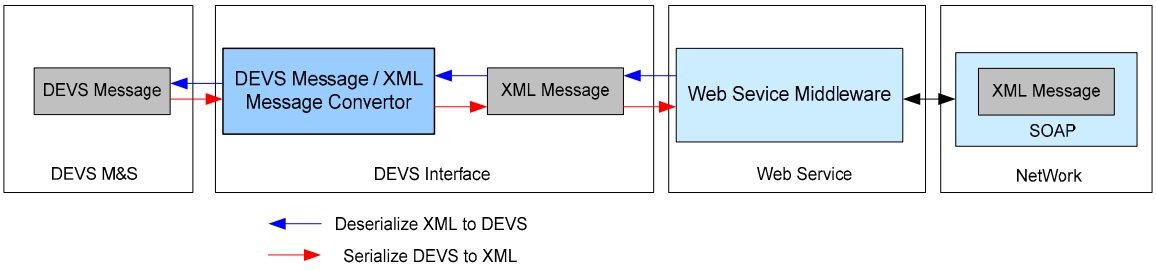

Figure 8: The DEVS message and XML message in the web service

Figure 8 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.

4 IMPLEMENTATION OF THE DEVS NAMESPACE AND DEVS SIMULATOR SERVICES

4.1 Implementation of the DEVS Namespace

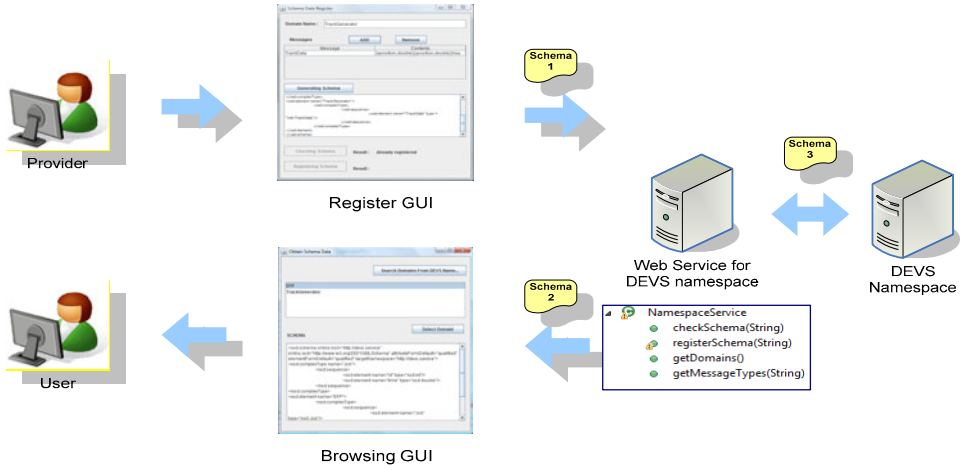

Figure 9: Overview of registering and browsing schema

We created a web service called *NamespaceService* through which schema of a DEVS simulator service is registered and browsed. Figure 9 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 storage, the user can use a browsing GUI consisting of two parts. One part is to display all schema documents in the DEVS namespace storage and the other part is to show the schema document corresponding to the name of the document chosen by the user.

4.1.1 The GUI for Schema Data Registration

The GUI has three functions: to enter message information such as class name, variable's name and type, to compose a schema document, and to check and register the schema to DEVS namespace storage. 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 storage. 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 namespace which represents a name of a DEVS model. Figure 10 represents the result of conversion "Job" message to a schema. The table on Figure 10 has two columns, *Message* and *Contents*, that display the messages. 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 pop up. As seen in the Figure 11, 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 10, we can see a schema document containing "EFP" namespace, "Job" class name and all names and types of variables in the "Job" class.

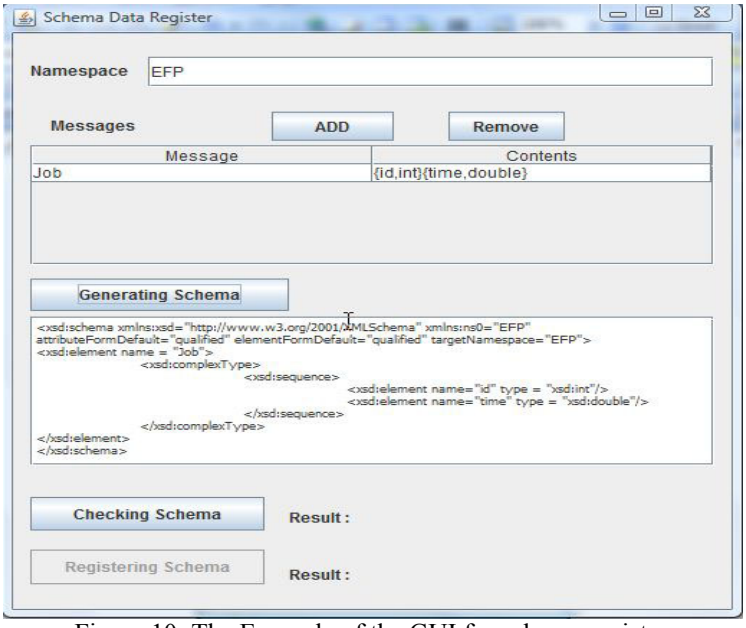

Figure 10: The Example of the GUI for schema register

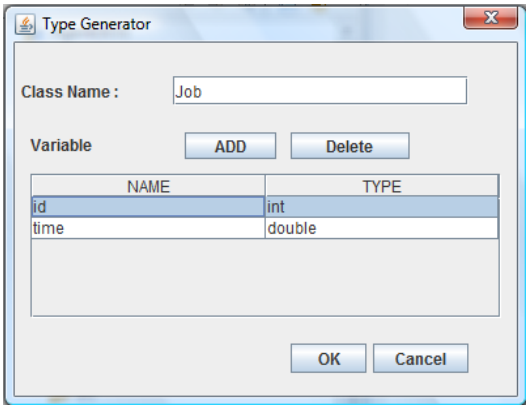

Figure 11: The GUI for type generator

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 storage. 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 *NamespaceService* Web Service

NamespaceService web service is designed to check, register, browse, and get a schema through 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 Implementation of the DEVS Simulator Service

In this paper, we use aDEVS and DEVSJAVA M&S with .Net and AXIS 2 environment, respectively. The procedure of generating the DEVS simulator service in the both environments requires common operations of the DEVS simulator service. Based on the operations, DEVS simulator services with .Net and AXIS 2 are created according to their own method of creating a web service. The details of how to create the DEVS simulator services with .Net and AXIS 2 are in Seo (2009). The DEVS simulator service with DEVSJAVA and AXIS 2 is briefly mentioned as follows.

To create web services for DEVSJAVA, we need packages such as DEVSJAVA API, DEVS interface, and a class containing operations of the DEVS simulator service. There are seven packages to create the DEVS simulator service (Seo 2009). 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.

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.

5 DEVS SIMULATOR SERVICES INTEGRATION AND EXECUTION

To demonstrate the DEVS simulator service interoperability system, an example DEVS model called *TrackCoupled* is used. The *TrackCoupled* has three atomic models called " Track Generator 1", "Track Generator 2", and "Track Display", respectively. The "Track Generator1" 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 namespace with a GUI for schema register. In this case, one message type, called *TrackData*, is used to send the track information

.

Table 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

```
<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 12: The schema for the TrackData

Table 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 12 represents the schema for the *TrackData* generated by the schema register GUI.

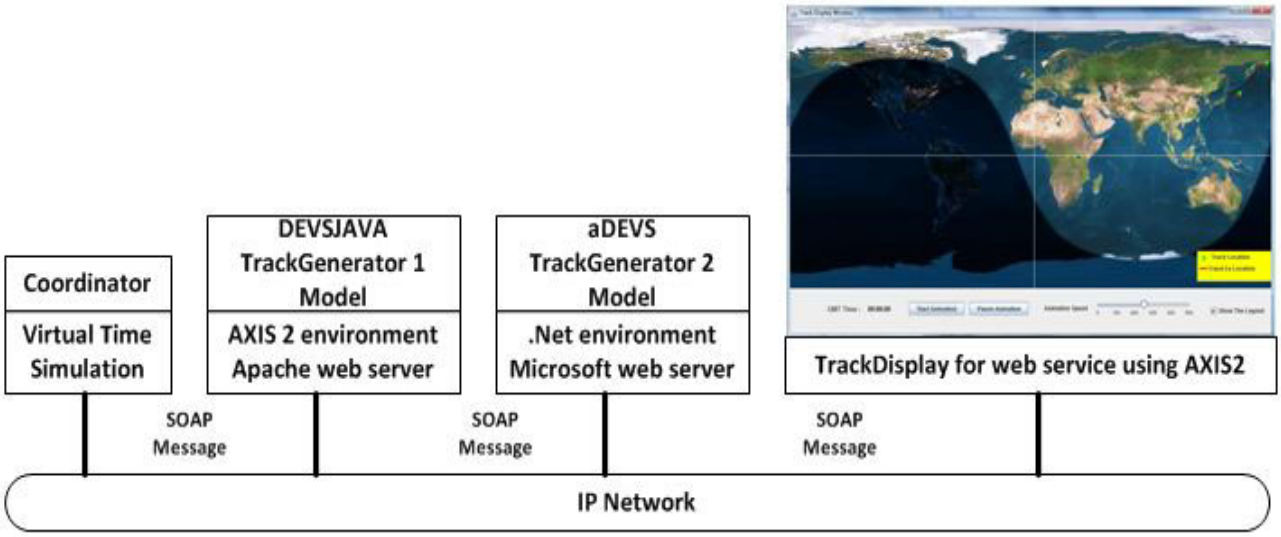

Figure 13: Simulation of DEVS simulator services

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. In the document, the locations of services and DEVS namespace and coupling information are displayed. 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. Figure 13 represents client and servers having services of atomic models in the *TrackCoupled*. When we execute the XML document, the track display window GUI is shown in the server containing the *TrackDisplay* service.

6 COMPARISON OF DEVS/SOA WITH HLA-BASED DEVS INTEROPERABILITY

Table 2: HLA vs. SOA support for DEVS interoperability

	DEVSsimHLA	**Interoperable DEVS/SOA**
Platform/Language interoperability	Support	Support
Neutral Message passing	No	Support
Middleware interoperability	Possible using a HLA bridge	Support
Linguistic levels of interoperability	Support 2 levels (syntactic and semantic)	Support all levels (syntactic, semantic, and pragmatic)

To point out the generic differences between interoperability supported by DEVS in HLA and in SOA, Table 2 presents a comparison between representative implementations, namely, DEVSsimHLA (Sung, Hong and Kim 2009) and DEVS/SOA, respectively. Both approaches support platform and language interoperability. However, while HLA uses pre-defined message types in the Federation Object Model (FOM) and Simulation Object Model (SOM), interoperable DEVS/SOA uses platform-neutral messages based on XML language during simulation. The neutral message passing enables heterogeneous DEVS models to be simulated with dynamic message invocation on the interoperable DEVS/SOA environment. Furthermore, it increases model reusability without changing structures of messages. From the middleware interoperability point of view, HLA implementations of different vendor can communicate with each other only using a HLA bridge. In contrast, DEVS simulators can interoperate without conflict because DEVS/SOA adheres to SOAP, the accepted standard for message exchange in web SOA.

7 CONCLUSION

In this study, we implemented an interoperable DEVS simulation environment adhering to the SOA and DEVS simulation standards, and extended with the DEVS namespace. In this environment, SOA provides network interoperability, DEVS protocol implementation provides simulator interoperability, and DEVS namespace provides a step toward semantic level interoperability. The DEVS simulator interoperability is implemented by DEVS simulator service consisting of three layers: 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 simulation results generated during the simulation period.

SOA uses an SOAP message to provide an interoperable environment. When SOA and DEVS meet, the DEVS message in its native language is converted to an XML DEVS message in compliance with the DEVS namespace. This XML DEVS message conforms to the DEVS formalism enabling interoperability over languages and platforms. Significantly, while in general, design of such marshalling/unmarshalling is manual, JAVA supports dynamic invocation enabling dynamic conversion of JAVA objects to XML messages. This invocation is implemented in the DEVS simulator services for DEVSJAVA.

The DEVS simulator level interoperability achieved so far paves the way for increasing the degree of semantic and pragmatic level interoperability in further development. Such increased capability can be facilitated by attaching metadata to models. For example, by associating the experimental frames in which models are developed to the models themselves, one can support increased semantic interoperability in the form of improved discoverability and appropriate reuse of models (Chreyh and Wainer 2009). Similarly, metadata can be attached to DEVS messages in the namespace to enhance the ability to find message schema that match the new model requirements. For example, the domain information currently employed to support namespace search can be augmented with experimental frame information that captures the objectives underlying the model development in which the DEVS messages originated. In this way, schema developed to serve objectives that are similar to those driving current model development stand a better chance of having the same meaning in the new context.

Pragmatic level interoperability requires shared agreements about the intentions of usage of the messages in the models. To support such interoperability requires that such representations enable applying messages that have been extracted from the namespace in a manner appropriate to the new context of use. Similar to the semantic level, pragmatic level interoperability can be increased by associating pragmatic frames with message schema in the namespace to enable the DEVS simulator integration service to match the schema to compatible usage contexts.

REFERENCES

Box, D., D. Ehnebuske, G. Kakivaya, and A. Layman. 2003. Simple Object Access Protocl (SOAP) 1.1.

Cheon, S., and B.P. Zeigler. 2006. Web Service Oriented Architecture for DEVS Model Retrieval by System Entity Structure and Segment Decomposition. DEVS Integrative M&S Symposium.

Cheon, S., C. Seo, S. Park, and B. P. Zeigler. 2004. Design and Implementation of Distributed DEVS Simulation in a Peer to Peer Network System. 2004 Advanced Simulation Technologies conference (ASTC '04) - Design, Analysis, and Simulation of Distributed Systems Symposium 2004 (DASD 2004).

Chreyh, R., and G. Wainer. 2009. CD++ Repository: An Internet Based Searchable Database of DEVS Models and Their Experimental Frames. DEVS Integrative M&S Symposium.

DiMario, M.J. 2006. 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.

Jacobs, R.W. 2004. Model-Driven Development of Command and Control Capabilities For Joint and Coalition Warfare, Command and Control Research and Technology Symposium.

Janousek, V., and E. Kironsky. 2006. Exploratory Modeling With SmallDEVS. In Proceedings of the 20th annual European Simulation and Modelling Conference.

Jarrah, M. 2008. An Automated Methodology for Negotiation Behaviors in Multi-Agent Engineering Applications. Electrical and Computer Engineering Dept., University of Arizona.

Kim, T., C. Seo, and B. P. Zeigler. 2008. Web Based Distributed SES/NZER Using Service Oriented Architecture. submitted to Simulation: Transactions of The Society for Modeling and Simulation International.

Kim, T. G., and J. Kim. 2005. DEVS Framework and Toolkits for Simulators Interoperation Using HLA/RTI. Asia Simulation Conference 2005, Bejing, China. 16 – 21.

Kuhl, F., R. Weatherly, and J. Dahmann. 1999. Creating Computer Simulation Systems: An Introduction to the High Level Architecture. Prentice Hall PTR.

Mittal, S., J. L. Risco-Martin, and B. P. Zeigler. 2007. DEVS-Based Simulation Web Services for Net-centric T&E. Summer Computer Simulation Conference SCSC'07.

Mittal, S., and J. L. R. Martín. 2007. 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*.

Newcomer, E., and G. Lomow. 2004. Understanding SOA with Web Services. Addison-Wesley Professional.

Nutaro, J.J. 2008. On Constructing Optimistic Simulation Algorithms for the Discrete Event System Specification. ACM Transactions on Modeling and Computer Simulation(TOMACS).

Sage, A. 2007. 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).

Sarjoughian, H. S., and B. P. Zeigler. 2000. DEVS and HLA: Complementary Paradigms for Modeling and Simulation. *Simulation: Transactions of the Society for Modeling and Simulation International 17*(4) : 187-97.

Seo, C., and B. P. Zeigler. 2009. Interoperability between DEVS Simulators using Service Oriented Architecture and DEVS Namespace. *A Joint Symposium DEVS Integrative M&S (DEVS) and High Performance Computing (HPC) Proceedings of the Spring Simulation Conference*

Seo, C., and B. P. Zeigler. 2009. Automating the DEVS Modeling and Simulation Interface to Web Services. *A Joint SymposiumDEVS Integrative M&S (DEVS) and High Performance Computing (HPC), Proceedings of the Spring Simulation Conference.*

Seo, C., S. Park, B. Kim, S. Cheon, and B. P. Zeigler. 2004. Implementation of Distributed high-performance DEVS Simulation Framework in the Grid Computing Environment. 2004Advanced Simulation Technologies conference (ASTC '04) -High Performance Computing Symposium 2004 (HPC 2004).

Sung, C. H., J. H. Hong and T. G. Kim. 2009. Interoperation of DEVS Models and Differential Equation Models using HLA/RTI: Hybrid Simulation of Engineering and Engagement Level Models. Proceedings of the DEVS Integrative M&S Symposium

Tolk, A., Saikou Y. Diallo, C. D. Turnitsa, and L. S. Winters. 2006. Composable M&S Web Services for Net-centric Applications. *Journal for Defense Modeling & Simulation (JDMS)*, 3(1): 27-44, January.

Touraille, L., M. K. Traoré, and D. R.C. Hill. 2009. A Mark-up Language for the Storage, Retrieval, Sharing and Interoperability of DEVS Models. DEVS Integrative M&S Symposium.

Vangheluwe, H., J. de Lara, J.-S. Bolduc, and E. Posse. 2001. DEVS Standardization: some thoughts. Winter Simulation Conference.

Wainer, G. 2002. CD++: a toolkit to develop devs models. Softw. Pract. Exper., 32(13):1261–1306.

Wutzler, T., and 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*.

Zeigler, B. P. 2004. DEVSJAVA 3.0. Available via <http://www.acims.arizona.edu/SOFTWARE/software.shtml#DEVSJAVA> [accessed August 3, 2009]

Zeigler, B.P., D. Fulton, P. Hammonds, and J. Nutaro, 2005. Framework for M&S Based System Development and Testing in Net-centric Environment. *ITEA Journal.* 26(3): 21-34.

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

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

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

AUTHOR BIOGRAPHIES

CHUNGMAN SEO is an senior research engineer in RTSync company and a member of Arizona Center for Integrative Modeling & Simulation (ACIMS). He received his Ph.D. in Electrical and Computer Engineering from The University of Arizona in 2009. His research interests include DEVS based web service integration; DEVS/SOA based distribution DEVS simulation, and DEVS simulator interoperability. His email address is <uracbul@gmail.com>.

BERNARD P. ZEIGLER is a Professor of Electrical and Computer Engineering at the University of Arizona, Tucson and Director of ACIMS. He is internationally known for his 1976 foundational text *Theory of Modeling and Simulation*, revised for a second edition (Academic Press, 2000), He has published numerous books and research publications on the Discrete Event System Specification (DEVS) formalism. In 1995, he was named Fellow of the IEEE in recognition of his contributions to the theory of discrete event simulation. His email address is <zeigler@ece.arizona.edu>.