

DEVSMML: Automating DEVS Execution Over SOA Towards Transparent Simulators

Saurabh Mittal*, José Luis Risco-Martín, Bernard P. Zeigler*
{saurabh, zeigler} @ece.arizona.edu, jlrisco@dacya.ucm.es

*Arizona Center for Integrative M&S
ECE Department, University of Arizona
Tucson, AZ 85721

Departamento de Arquitectura de Computadores y
Automática
Universidad Complutense de Madrid
28040 Madrid, Spain

Keywords:

DEVSMML, SOA, Web services, JavaML

Abstract

Discrete Event Specification (DEVS) formalism has been used to study dynamics of discrete event systems. DEVS environments are typically open architectures that have been extended to execute on various middleware such as CORBA, Grid computing, P2P networks, RMI and others. The present work aims to provide another development environment using the Service Oriented Architecture (SOA) framework. The proposed DEVS Modeling Language (DEVSMML) is built on XML and provides model interoperability among DEVS models located at remote locations. The DEVSMML environment is built on client-server paradigm and the simulation is executed at the server's end. The proposed DEVS atomic and coupled DTDs are open to standardization from the community for successful model sharing and collaboration. The DEVSMML framework provides the needed feature of run-time composability of coupled systems using the SOA framework. DEVSMML also provides the capability to translate model to and from XML and JAVA programming language leading to model composability and validation. This paper will demonstrate the client application as well as the server architecture underlying the DEVSMML framework.

1. Introduction

DEVS formalism [1] exists in many implementations, primarily in DEVS/C++ and DEVSMML [2]. Extensions of these implementations are available as DEVS/HLA [3], DEVS/CORBA [4], cell-DEVS [5], and DEVS/RMI [6]. Since DEVS is inherently based on object oriented methodology, C++ and Java are the chosen programming languages. Almost all of the extensions capitalize on the underlying object orientation provided by these two programming languages. The models are coded either in C++ or Java. DEVS formalism categorically separates the model, the Simulator and the Experimental frame. However, one of the major problems in this kind of mutually exclusively system is that the formalism implementation is itself limited by the underlying programming language. In other words, the model and the simulator exist in the same programming language. Consequently, legacy models as well as models that are available in one implementation are hard to translate from one language to another even though both the implementations are

object oriented. Other constraints like libraries inherent in C++ and Java are another source of bottleneck that prevents such interoperability.

The motivation for this work stems from this need of model interoperability between the disparate simulator implementations and provides a means to make the simulator transparent to model execution. We propose DEVS Modeling Language (DEVSMML) that is built on eXtensible Markup Language (XML) [7] as the preferred means to provide such transparent simulator implementation. The present work has been done with Java and efforts are ongoing in the direction to provide C++ implementation of the concept. This work is built on the JAVAML research done by Vladimir for DEVS Meta Language [8]. While his work aims to provide a stand-alone XML schema for DEVS formalism that can be used by any of programming implementations, research is still ongoing to specify the logic behavior in atomic models. The present work aims to extend his approach and provide complete behavioral support in DEVSMML by implementing the proposed universal Atomic and Coupled DTDs. We look forward toward standardization of these DTDs so that models across the web can participate in Dynamic Modeling & Simulation over Net-centric web services.

We have implemented our proposed DTDs in web service architecture; specifically a Service Oriented Architecture (SOA) [9] and paper will illustrate the Server as well as Client designs. We also propose modifications in the DEVS formalism as well that will make a DEVS model to be a DEVS Service model that can be readily deployed using Model-continuity principles [10].

The paper is organized as follows. The next section provides information about the related work. Section 3 provides basic information about the underlying technologies for the development of DEVSMML SOA framework. Section 4 provides an overview of DEVSMML layered architecture. Section 5 provides detailed DEVS DTDs. Section 6 presents the Web Service Architecture with both Server and Client designs. Section 7 demonstrates how a client can use the DEVSMML transparent simulator implementation and compose models with existing remote models. Section 8 provides conclusion and the recommended future work.

2. Related Work

There have been a lot of efforts in the area of distributed simulation using parallelized DEVS formalism. Issues like ‘causal dependency’ [1] and ‘synchronization problem’ [11] have been adequately dealt with solutions like: 1. restriction of global simulation clock until all the models are in sync, or 2. rolling back the simulation of the model that has resulted in the causality error. Our chosen method of web centric simulation does not address these problems as they fall in a different domain. In our proposed work, the simulation engine rests solely on the Server. Consequently, the coordinator and the model simulators are always in sync.

Most of the existing web-centric simulation efforts consist of the following components:

1. *the Application*: the top level coupled model with (optional) integrated visualization.
2. *Model partitioner*: Element that partitions the model into various smaller coupled models to be executed at a different remote location
3. *Model deployer*: Element that deployed the smaller partitioned models to different locations
4. *Model initializer*: Element that initializes the partitioned model and make it ready for simulation
5. *Model Simulator*: Element that coordinate with root coordinator about the execution of partitioned model execution.

The Model Simulator design is almost same in all of the implementation and is derived directly from parallel DEVS formalism [1]. There are however, different methods to implement the former four elements. DEVS/Grid [12] uses all the components above. DEVS/P2P [13] implements step 2 using hierarchical model partitioning based on cost-based metric. DEVS/RMI [6] has a configuring engine that integrates the functionality of step 1, 2 and 3 above. DEVS/Cluster [14] is a multi-threaded distributed DEVS simulator built on CORBA, which again, is focused towards development of simulation engine.

As stated earlier, the efforts have been in the area of using the parallel DEVS and implementing the simulator engine in the same language as that of the model. Our present work is not focused in this area. It is focused towards interoperability at the application level, specifically, at the model level and hiding the simulator engine as a whole.

The research of DEVS Standardization group [15] can be divided into four basic areas [8]:

1. Standardization of DEVS formalism
2. Standardization of DEVS models
3. Standardization of the interface of DEVS Simulator
4. Standardization of libraries of DEVS models

Members of Standardization group have worked concerning area 2 where the model’s structure is based

on XML [16, 17]. However, their general modeling tool ATOM3 [17] is based on meta-meta-modeling. It is based on graph grammars and allows transformation of model to different formalism. Vladimir’s [8] work is concerning areas 2 and 4. His implementation of DEVS meta model is based on underlying JAVA Modeling Language (JAVAML) [18]. Vladimir presents a prototype of a modeling tool that aims towards model interoperability but the paper lacks sufficient details and any working example. Our earlier work presents the detailed W3C Schema for DEVS atomic and coupled models [19] as intended by Vladimir. Other research effort using XML description is done by [20] called as DEVSW fits areas 2 and 4 but the code for transition functions is provided by means of pseudo code.

These efforts are in no means similar to what we are proposing in our paper, except some of ideas presented by Vladimir. The mentioned efforts are aimed towards development of an independent meta-language that would aid the user to write models effectively and easily and then the process of model generation and simulation is automated using XML. We are focused towards taking XML just as a communication middleware, as used in SOAP, for existing DEVS models. We would like the user or designer to code the behavior in any of the programming languages and let the DEVSMIL SOA architecture be responsible to create a coupled model, integrating code in either of the languages and delivering us with an executable model that can be simulated. The user need not learn any new syntax, any new language; however, what he must use is the standardized version of DEVS implementation such as DEVJSJAVA Version 3.0 [2] (maintained at www.acims.arizona.edu).

This kind of capability where the user can integrate his model from models stored in any web repository, whether it contained public models of legacy systems or proprietary standardized models will provide more benefit to the industry as well as to the user, thereby truly realizing the model-reuse paradigm.

Our work spans areas 2, 3, and 4. In further sections we will provide details about DEVS atomic and coupled DTDs, design of DEVS Simulator interface and standardized libraries used in our implementation.

3. Underlying Technologies

3.1 DEVS

DEVS formalism consists of models, the simulator and the Experimental Frame. We will focus our attention to the two types of models i.e. atomic and coupled models. The atomic model is the irreducible model definitions that specify the behavior for any modeled entity. The coupled model is the aggregation/composition of two or more atomic models connected by explicit couplings.

The formal definition of P-DEVS atomic model in parallel DEVS is given in [1]:

$$\mathbf{M} = \langle \mathbf{X}, \mathbf{S}, \mathbf{Y}, \delta_{\text{int}}, \delta_{\text{ext}}, \delta_{\text{conf}}, \lambda, \text{ta} \rangle$$

Where

\mathbf{X} is the set of input values;

\mathbf{S} is the state space

\mathbf{Y} is the set of output values;

$\delta_{\text{int}}: \mathbf{S} \rightarrow \mathbf{S}$ is the internal transition function

$\delta_{\text{ext}}: \mathbf{Q} \times \mathbf{X}^b \rightarrow \mathbf{S}$ is the external transition function, where \mathbf{X}^b is a set of bags over elements in \mathbf{X} ,

$$\delta_{\text{ext}}(s, e, \emptyset) = (s, e)$$

$\mathbf{Q} = \{(s, e) \mid s \in \mathbf{S}, 0 \leq e \leq \text{ta}(s)\}$ is the total state set
 e is the time elapses since last transition

$\delta_{\text{conf}}: \mathbf{S} \times \mathbf{X}^b \rightarrow \mathbf{S}$ is the confluent transition function

$\lambda: \mathbf{S} \rightarrow \mathbf{Y}$ is the output function

$\text{ta}: \mathbf{S} \rightarrow \mathbf{R}_{0 \rightarrow \infty}$ is the time advance function

The formal definition of a coupled model is described as:

$$\mathbf{N} = \langle \mathbf{X}_{\text{self}}, \mathbf{Y}_{\text{self}}, \mathbf{D}, \mathbf{EIC}, \mathbf{EOC}, \mathbf{IC} \rangle$$

where,

\mathbf{X}_{self} is the set of external input events,

\mathbf{Y}_{self} is the set of output events,

\mathbf{D} is a set of DEVS component models

\mathbf{EIC} is the external input coupling relation

\mathbf{EOC} is the external output coupling relation

\mathbf{IC} is the internal coupling relation

The coupled model \mathbf{N} can itself be a part of component in a larger coupled model system giving rise to a hierarchical DEVS model construction. Detailed descriptions about DEVS Simulator, Experimental Frame and of both atomic and coupled models can be found in [1].

3.2 Web Services and Interoperability using XML

Service oriented Architecture (SOA) framework is a framework consisting of various W3C standards, in which various computational components are made available as ‘services’ interacting in an automated manner towards achieving machine-to-machine interoperable interaction over the network. The interface is specified using Web Service Description language (WSDL) [21] that contains information about ports, message types, port types, and other relating information for binding two interactions. It is essentially a client server framework, wherein client request a ‘service’ using SOAP message that is transmitted via HTTP in XML format. A Web service is published by any commercial vendor at a specific URL to be consumed/requested by another commercial application on the Internet. It is designed specifically for machine-to-machine interaction. Both the client and the server encapsulate their message in a SOAP wrapper.

3.3 JavaML

JavaML [18] is an XML-Based source code representation for Java programs. The JAVAML Document Type Definition (DTD) specifies various elements of a valid JavaML document. It is well-suited

to be used as canonical representation of Java source code for tools. It comes with an XSLT-based back-converter that translates a JavaML document back into java source code. More details about JavaML can be found at [18].

4. Overview of DEVSML

DEVSML is a novel way of writing DEVS models in XML language. This DEVSML is built on JAVAML, which is infact, XML implementation of JAVA. The current development effort of DEVSML takes its power from the underlying JAVAML that is needed to specify the ‘behavior’ logic of atomic and coupled models. The DEVSML models are transformable back’n forth to java and to DEVSML. It is an attempt to provide interoperability between various models and create dynamic scenarios. The key concept is shown in the figure 1 below:

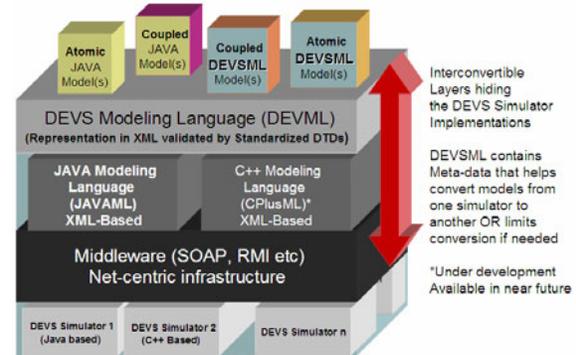


Figure 1: DEVS Transparency and Net-centric model interoperability using DEVSML

The layered architecture of the said capability is shown in Figure 1. At the top is the application layer that contains model in DEVS/JAVA or DEVSML. The second layer is the DEVSML layer itself that provides seamless integration, composition and dynamic scenario construction resulting in portable models in DEVSML that are complete in every respect. These DEVSML models can be ported to any remote location using the net-centric infrastructure and be executed at any remote location. Another major advantage of such capability is total simulator ‘transparency’. The simulation engine is totally transparent to model execution over the net-centric infrastructure. The DEVSML model description files in XML contains meta-data information about its compliance with various simulation ‘builds’ or versions to provide true interoperability between various simulator engine implementations. This has been achieved for at least two independent simulation engines as they have an underlying DEVS protocol to adhere to. This has been made possible with the implementation of a single atomic DTD and a single coupled DTD that validates the DEVSML descriptions generated from these two implementations. Such run-time interoperability

provides great advantage when models from different repositories are used to compose bigger coupled models using DEVSML seamless integration capabilities.

Figure 2 provides a basic flow chart of operations that can be done with DEVSML framework. The designer can start with either the JAVA code for atomic/coupled model or the DEVSML code for atomic/coupled model. In either of the case, the process has to lead to DEVSML representation of the model. The DEVSML description that is essentially XML is then validated by the standardized DTDs (shown in next section), can now

participate in model composition (blue box). The composed coupled model as well as DEVSML atomic model can verily be stored in the Library for reuse. The composed integrated model, that is complete in every respect, as it contains behavior as well, as ready for simulation. The DEVSML model is then sent to various remote locations or specifically Server, wrapped in SOAP message to the destination host (Server in our case). Based on the information contained in the DEVSML model description, corresponding simulator is called for to instantiate the model and executes the simulation with the designated simulator.

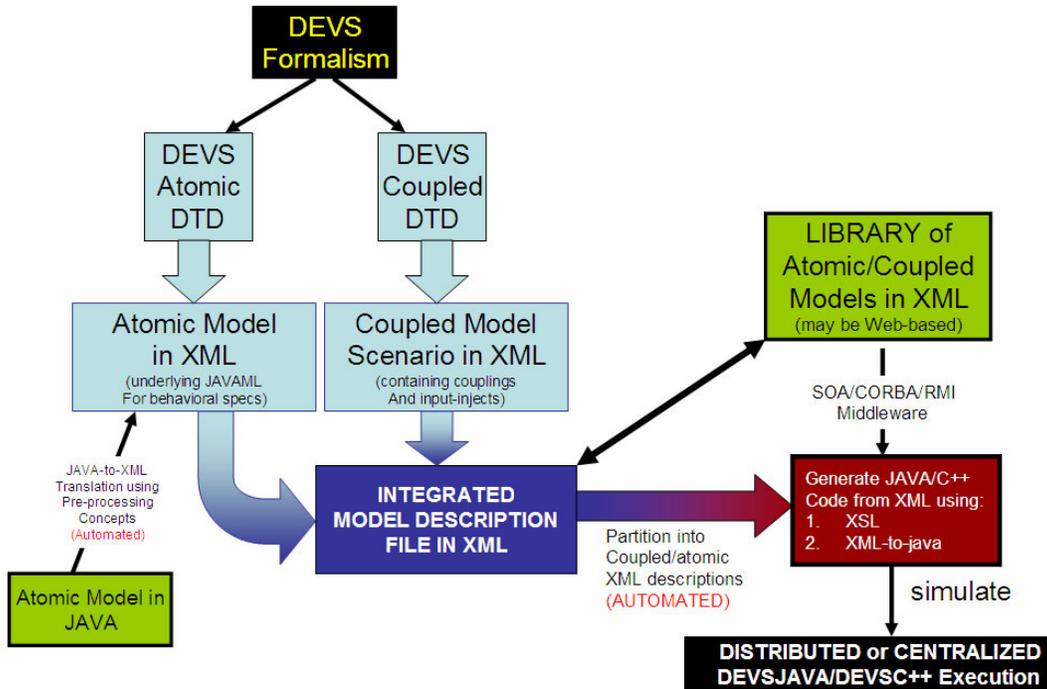


Figure 2: Flow chart of basic operations leading to model composability using DEVSML

5. DEVS DTDs and their Standardization

This section provides details about the modified DEVS formalism for the atomic model to make it ‘service enabled’ in the process of software engineering. The motivation comes from the fact that testing of Web Services as in ‘system test suite’ is still in infancy and DEVS based testing is still in progress. With a slight modification in the DEVS formalism for atomic model we plan to achieve the following:

1. Transform any existing DEVS atomic as a container that is capable of publishing services
2. Promote testing of web service components by making them DEVS enable so that a DEVS wrapper would encapsulate a Service as a ‘component’
3. Transition from a DEVS Service component directly to a web service component after removal of wrapper and deploy it using model-continuity principles.

Figure 3 provides a graphical view of an abstract component that inherits the basic functionality of DEVS atomic model. The extended DEVS formalism is specified as below:

$$SM = \langle X, S, Y, \delta_{int}, \delta_{ext}, \delta_{conf}, \lambda, ta, V \rangle$$

where,

V is the set of Service methods that are represented by this atomic model.

The other symbols have their usual meaning as described in section 3.1.

The XML representation of this abstract component is shown in Figure 4. The idea here is that a DEVS atomic model contains the behavior of a component that has defined interfaces. The *devsObject* is the wrapper that takes care of δ_{int} , δ_{ext} , δ_{conf} interactions, while the *serviceObject* presents the services, or methods that are either used individually or in nested manner to

implement a published service. Making this change in the DEVS formalism does not change DEVS original formalism. It just introduces a container that contains the name of the methods that could be published as a service. In complex models, it is a common practice to break the use-case into smaller manageable use-cases for implementation purposes. Similarly, implementing complex behaviors and complicated state machines (Mitt06) require the functionality to be organized into methods that are called in the DEVS δ_{int} , δ_{ext} , δ_{conf} functions. The set V keeps an account of such methods that can be made available for service publications.

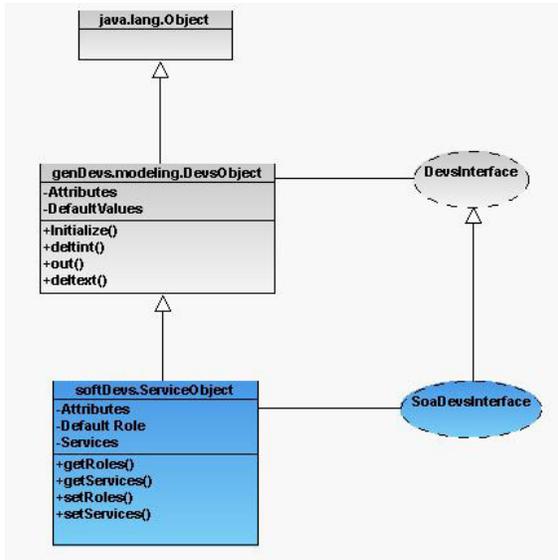


Figure 3: an SOA object capable of DEVS modeling

As shown in Figure 4 below, the XML structure of a *serviceObject* is implemented based on the UML diagram in Figure 3. What is required here is the addition of code for ‘services’ tag. Once implemented on SOA, the code with respect to the ‘services’ tag can be exchanged through a SOAP message and a DEVS model is made ready for simulation.

Figure 5 shows the DEVSML DTD for extended DEVS formalism that contains the ‘services’ container. Similarly, Figure 6 contains the DTD for DEVS coupled (digraph) model. The coupled model is a hierarchical model that takes into account of the contained atomic or coupled models. Also notice the attribute ‘**simulator (devsjava|xdevs)**’ in the ATTLIST tag for atomic as well as coupled element. This is the meta-data that is stored with every model that is used by server to assign the appropriate simulator for this model. Components within a coupled model could be managed by different simulators. The attribute simulator in the nodes *coupledRef* and *atomicRef* (see Figure 6) defines the simulator to use. This attribute is generated when de whole model is integrated in one DEVSML file. Of course, the simulator must comply with the DEVS simulation protocol. The authors call for standardization of both of these DTDs.

```

<?xml version="1.0" encoding="UTF-8"?>
<xml-body>
<model>
  <atomic>
    <name>Hello</name>
    <params> </params>
    <construct>
      <args> </args>
      <ports>
        <inports>
          <inport>in</inport>
        </inports>
        <outports>
          <outport>out</outport>
        </outports>
      </ports>
    </construct>

    <initialize>
    </initialize>
    . . .

  </atomic>
</model>
</xml-body>

<services>
  <function>
    <access> public </access>
    <return> int </return>
    <inport> in </inport>
    <outport> out </outport>
    <fname> decrement() </fname>
    <logic> </logic>
  </function>
</services>
</atomic>
</model>
</xml-body>
  
```

Figure 4: Automated XML snippet for a DEVS atomic model.

```

<!-- DEVS ATOMIC MODEL -->
<!ENTITY % variable-info
  "name CDATA #REQUIRED
  type CDATA #REQUIRED">
<!ELEMENT atomic
  (inputs,outputs,states,ta,deltint,delttext,deltcon,lambda,services?,java-specific?)>
<!ATTLIST atomic
  name ID #REQUIRED
  simulator (devsjava|xdevs) #REQUIRED
  host CDATA #REQUIRED>
<!ELEMENT inputs (port*)>
<!ELEMENT port EMPTY>
<!ATTLIST port
  name CDATA #REQUIRED>
<!ELEMENT states (state*)>
<!ELEMENT state EMPTY>
<!ATTLIST state
  %variable-info;>
<!ELEMENT outputs (port*)>
<!ELEMENT ta (block?)>
<!ELEMENT deltint (block?)>
<!ELEMENT delttext (block?)>
<!ELEMENT deltcon (block?)>
<!ELEMENT lambda (block?)>
<!ELEMENT services (service*)>
<!ELEMENT service (method)>
<!ATTLIST service
  name ID #REQUIRED
  port CDATA #REQUIRED>
<!ELEMENT java-specific (package-decl,import*,constructor*,method*)>
<!ELEMENT import EMPTY>
  
```

Figure 5: DEVS atomic DTD

```

<!--DEVS COUPLED MODEL-->
<!ENTITY % connection-info
  "component_from CDATA #REQUIRED
  port_from CDATA #REQUIRED
  
```

```

        component_to CDATA #REQUIRED
        port_to CDATA #REQUIRED">
<!ELEMENT devs (scenario,models)>
<!ELEMENT scenario (coupled)>
<!ELEMENT coupled
(inputs,outputs,components,internal_connections
,external_input_connections,external_output_con
nections,java-source-program)>
<!ATTLIST coupled
name ID #REQUIRED
model CDATA #REQUIRED
simulator (devsjava|xdevs) #REQUIRED
host CDATA #REQUIRED>
<!ELEMENT inputs (port*)>
<!ELEMENT port EMPTY>
<!ATTLIST port
name CDATA #REQUIRED>
<!ELEMENT outputs (port*)>
<!ELEMENT components (coupledRef|atomicRef)*>
<!ELEMENT coupledRef (components?)>
<!ATTLIST coupledRef
name CDATA #REQUIRED
model CDATA #REQUIRED
simulator (devsjava|xdevs) #IMPLIED
host CDATA #REQUIRED>
<!ELEMENT atomicRef EMPTY>
<!ATTLIST atomicRef
name CDATA #REQUIRED
model CDATA #REQUIRED
simulator (devsjava|xdevs) #IMPLIED
host CDATA #REQUIRED>
<!ELEMENT internal_connections (connection*)>
<!ELEMENT external_input_connections
(connection*)>
<!ELEMENT external_output_connections
(connection*)>
<!ELEMENT connection EMPTY>
<!ATTLIST connection
%connection-info;>

<!ELEMENT models (model*)>
<!ELEMENT model (java-source-program)>
<!ATTLIST model
name ID #REQUIRED>

```

Figure 6: DEVS coupled DTD

6. Web Services Architecture for DEVSMML

Figure 7 shows the designed Web Architecture. At server's end, there are N simulators registered, and the WSDL files containing the Web services offered and an Applet for generation and simulation of DEVSMML models that uses these Web services. At the client's end, it is possible to use the Applet or an own client program [22], which makes use of the Web services (in Figure 7: CLAPP, Client Application).

Registering a simulator means to enable it so that it can be used according to the defined DEVSMML DTDs. This involves the definition of two additional classes that implement the interfaces *InterfaceXmlAtomic* and *InterfaceXmlCoupled* (see Figure 7). These classes must generate XML *elements* that define the structure of the specific simulator models according to the atomic and coupled DTDs, These *elements* are inputs, outputs, etc. Efforts are ongoing to develop a template for the user community to register their simulators via a new process in order to make the registration process easier.

Once the simulator is registered, the Web services are available for this simulator. The registry is recommended, since the clients can use any simulator registered at the server.

The most important Web services offered in our current architecture are:

- Convert Java models to DEVSMML.
- Convert DEVSMML models to Java.
- Integrate coupled and atomic DEVSMML models towards a portable 'Composite' Coupled DEVSMML file that can be simulated at any server.
- Validate an existing DEVSMML model.
- Simulate a Composite Coupled file at the server.

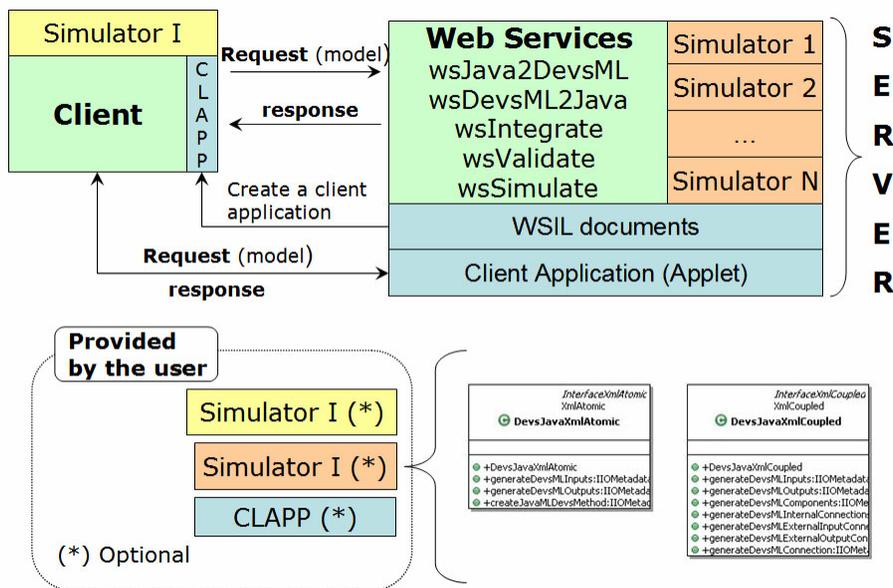


Figure 7: Web service Architecture for DEVSMML Implementation

Figure 8 shows part of the UML diagram of the Applet developed. *xdevs* and *devsjava* classes are directly generated from the Web services since we have these two simulators registered. The rest of the diagram provides the functionality of the Applet. Providing complete details is outside the scope of this article and will be report in our forthcoming publication dedicated to Server and Client designs. Demonstration of these web services is available at [22] that is hosted at ACIMS www.acims.arizona.edu.

Systems M&S based on DEVS theory [1] and web-based collaborative modeling leading to composite coupled models based on DEVSMML has been attempted for the Java programming language. In order to solve the same problem for other programming languages such as C++, C#, ADA, etc., we can choose among different alternatives:

- Using JNI. In this case, it is necessary to adapt each simulator to JNI. Therefore, the models must be rewritten into Java. The reason behind this conversion is due to the fact that we need behavior representation in XML. We do have cppML, that is C++ Modeling Language in XML but we want only one behavioral representation in XML. Our preferred way of doing it is through JavaML as Java is better positioned to address the Web Services domain.

- Using another XML representation more versatile for the behavior of the model. In this case it is possible to use XML definitions defined to represent any object oriented programming language, such as o:XML [23] or OOPML [24]. This is again a work in progress.

The disadvantage of using one solution or other resides in the interoperability between different simulators executing the same model. Proving interoperability between simulators is what true transparency is. If all the simulators are running under JNI, then adapters must be made in order to change information among them. The current DEVSMML architecture with only one universal underlying atomic DTD and coupled DTD is the first step towards interoperable simulators. Defining a distributed coordinator between these simulators is the second step. If o:XML or OOPML are used, then it is not necessary to define JNI simulators or rewrite models in Java, but what is needed a mechanism to interoperate between different DEVS simulators. How to communicate a simulator written in C++ with a simulator written in Java? Perhaps the solution resides in the definition of standards for the format of the data at the syntactic level.

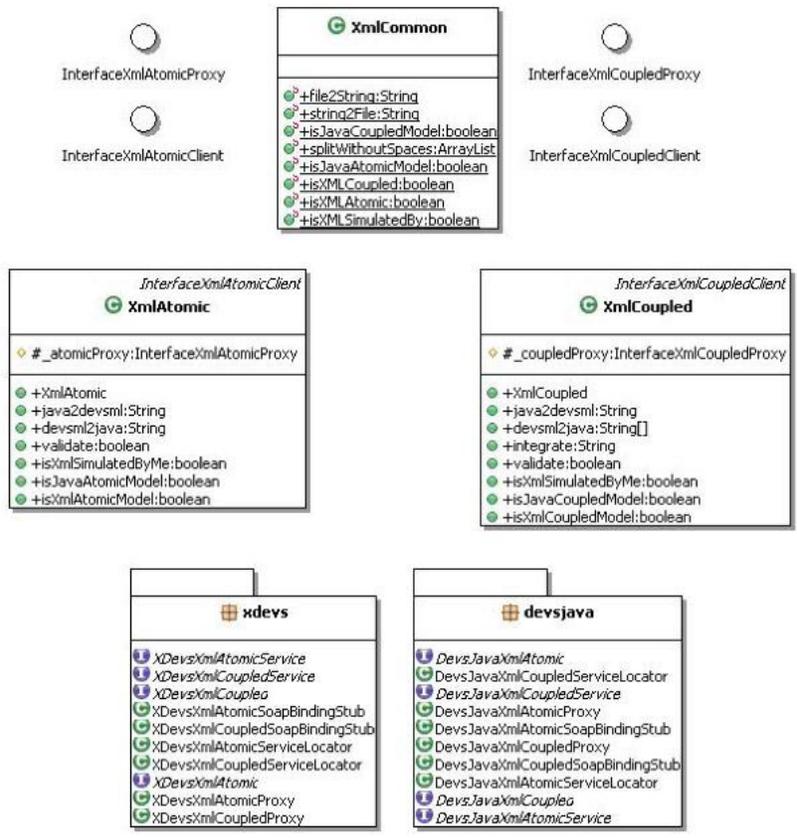


Figure 8: Client side implementation using interfaces.

7. DEVSML Application Development

This section provides information about the client application that communicates with the server resting at both ACIMS center and at Spain (redundancy purposes). The application is made available as an applet [22] or as a .exe application that is capable of communicating to the server at client's end.

The following snapshot shows the java application Ver. 2.0 that demonstrates the following:

1. Contains two simulator operability i.e xDEVs (Spain) [25] and GenDEVs (ACIMS-USA) [2] demonstrating validation of DEVSML atomic and coupled models with same Atomic and Coupled DTD
2. Converts any atomic/coupled model from their JAVA implementation to DEVSML transformation and vice-versa
3. Validates any DEVSML model description
4. Integrates any coupled DEVSML description into a composite DEVSML coupled model ready to be simulated with corresponding simulator

5. Generation of JAVA code library from a composite DEVSML coupled model.

6. Out of ten web services in operation, five Web Services that are publicly offered are:

- a. Convert Java model to DEVSML
- b. Convert DEVSML to java code
- c. Validate the existing DEVSML model
- d. Integrate coupled and atomic DEVSML models

towards a portable 'Composite' Coupled DEVSML file that is Simulatable at any remote server

e. Simulates the Composite Coupled file and sends console messages at Server to Client window giving evidence of simulation running.

7. Server rests at ACIMS lab that provides these Services

8. User can select his own Source and Target directories
9. User can choose his chosen implementation i.e. java code and Simulator compatibility. The Server application checks for compatibility as well.

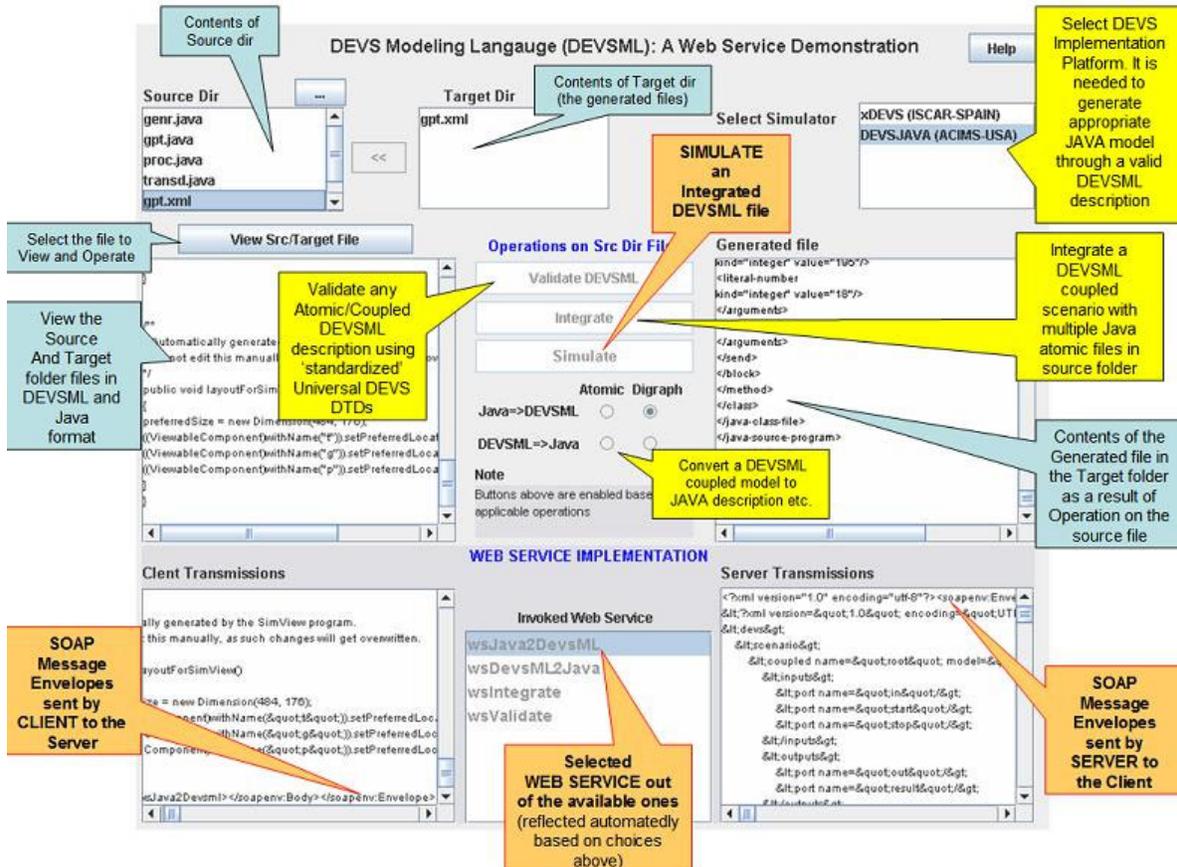


Figure 10: Client application snapshot implemented as an applet.

8. Conclusions and Future Work

We have addressed the problem of model interoperability with a novel approach of developing DEVSML as the transformation medium towards composability and dynamic scenario construction. The

composed coupled models are then validated using the proposed universal atomic and coupled DTDs. The simulators validated at the server's end are maintained centrally such that the efforts of the community can be brought together through the standardized processes. Other advantage of using DEVSML as the

communication medium gives the coder the independence to concentrate on the behavior of the component in their native languages (C++ and Java). In addition, it gives them the capability to share and integrate their models with that of other remote models and get that integrated validated model back in their own language. It also gives models the capability to get simulated with various simulator implementations that are stored at Server. This information is stored in meta-data that is contained in every model. Currently, this capability is meant only for Java but efforts are in progress to develop the corresponding methodology in C++ and will be reported in future. The paper also proposes modification in DEVS formalism towards making them Service capable such that model continuity can be exploited towards deploying any DEVS component as a Service. Related efforts in distributed simulation that employed concepts like model-partitioning and automated model-deployment are other areas that needs some work under DEVSMIL implementation. The idea is to have multiple servers running as a cluster and model being simulated without any knowledge of simulator coupling at its own end.

References:

- [1] Zeigler, B., Kim, T., Praehofer, H., *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press, 2000
- [2] ACIMS software site:
<http://www.acims.arizona.edu/SOFTWARE/software.shtml> Last accessed Nov 2006
- [3] Sarjoughian, H.S., B.P. Zeigler, "DEVS and HLA: Complimentary Paradigms for M&S?" Transactions of the SCS, (17), 4, pp. 187-197, 2000
- [4] Cho, Y., B.P. Zeigler, H.S. Sarjoughian, Design and Implementation of Distributed Real-Time DEVS/CORBA, IEEE Sys. Man. Cyber. Conf., Tucson, Oct. 2001.
- [5] 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
- [6] Zhang, M., Zeigler, B.P., Hammonds, P., DEVS/RMI-An Auto-Adaptive and Reconfigurable Distributed Simulation Environment for Engineering Studies, ITEA Journal, July 2005
- [7] XML: <http://www.w3.org/XML/>
- [8] Janousek, V., Polasek, P., Slavicek, P., Towards DEVS Meta Language, In ISC Proceedings, Zwinjnaarde, BE 2006, p 69-73 ISBN-90-77381-26-0
- [9] <http://java.sun.com/developer/technicalArticles/WebServices/soa/>
- [10] Hu X., A Simulation Based Software Development Methodology for Distributed Real-time Systems, PhD Dissertation, Fall 2003
- [11] Fujimoto, R.M., *Parallel and Distribution Simulation Systems*, Wiley, 1999
- [12] Seo, C., Park, S., Kim, B., Cheon, S., Zeigler, B.P., Implementation of Distributed High-performance DEVS Simulation Framework in the Grid Computing Environment, Advanced Simulation Technologies conference (ASTC), Arlington, VA, 2004
- [13] Cheon, S., Seo, C., Park, S., Zeigler, B.P., Design and Implementation of Distributed DEVS Simulation in a Peer to Peer Networked System, Advanced Simulation Technologies Conference, Arlington, VA, 2004
- [14] Kim, K., Kang, W., CORBA-Based, Multi-threaded Distributed Simulation of Hierarchical DEVS Models: Transforming Model Structure into a Non-hierarchical One, International Conference on Computational Science and Its Applications, ICCSA, Italy 2004
- [15] Vangheluwe, H., Bolduc, L., Posse, E. DEVS Standardization: some thoughts, Winter Simulation Conference 2001
- [16] Fishwick, P., XML Based Modeling and Simulation Using XML for Simulation Modeling, Proceedings of the 34th conference on Winter Simulation; exploring new frontiers, pg. 616-622, 2002
- [17] Lara, J., Vangheluwe, H., AToM3 as a Meta-CASE environment (DFD to SC), 4th International Conference on Enterprise Information Systems 2002
- [18] 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
- [19] Martin, JLR, Mittal, S., Pena, MAL, Cruz, JM., A W3C XML Schema for DEVS Scenarios, submitted to DEVS Symposium 2006
- [20] Yung-Hsim, W., Yao-Chung, L., A Modeling and Simulation Example Using DEVSW, 2002
- [21] WSDL <http://www.w3.org/TR/wSDL>
- [22] DEVSMIL – A Web Service Demonstration
<http://150.135.218.205:8080/devsmil/>
- [23] o:XML: www.o-xml.org
- [24] OOPML: <http://xml.coverpages.org/oopml.html>
- [25] XDEVs web page:
<http://itis.cesfepipesegundo.com/~jlrisc/xdevs.html>